

@notaciones



Manual de Uso Avanzado

Título: Anotaciones Java. Manual de Uso Avanzado.

Autor: Miguel Ángel Rodríguez Manzano .

Versión: 1.1 (5 de Diciembre de 2017).

Director: Sergio Gálvez Rojas.

Titulación: Ingeniero en Informática.

Departamento: Lenguajes y Ciencias de la Computación.

Centro: Escuela Técnica Superior de Ingeniería Informática.

Universidad de Málaga.



ÍNDICE DE CONTENIDOS

1.- ACERCA DEL MANUAL.....	7
1.1.- Estructuración del contenido.....	7
1.2.- Itinerarios de lectura.....	8
1.3.- Anexos de código fuente.....	8
2.- ORIGEN DE LAS ANOTACIONES.....	9
2.1.- Creación de la plataforma Java.....	9
2.2.- Evolución de la plataforma Java hasta J2SE 1.5.....	10
2.3.- Anotaciones en J2SE 1.5.....	11
3.- CONCEPTOS BÁSICOS.....	12
3.1.- Concepto de anotación.....	12
3.2.- Procesamiento de anotaciones.....	13
3.3.- Usos habituales de las anotaciones.....	14
4.- ANOTACIONES.....	15
4.1.- Sintaxis.....	15
4.2.- Variantes sintácticas.....	15
4.2.1.- Anotación normal.....	15
4.2.2.- Anotación marcadora (Marker Annotation).....	16
4.2.3.- Anotación sencilla (Single Element Annotation).....	16
4.3.- Elementos de código anotables.....	17
4.4.- Ubicación de las anotaciones.....	18
4.5.- Número de anotaciones.....	19
4.6.- Anotaciones repetibles (Java SE 8+).....	20
5.- TIPOS ANOTACIÓN.....	22
5.1.- Sintaxis.....	22
5.2.- Tipos permitidos para los elementos.....	23
5.3.- Valores permitidos y por defecto en elementos.....	24
5.4.- Meta-anotaciones sobre los tipos anotación.....	24
5.5.- Cuadro de referencia de un tipo anotación.....	25
5.6.- Ejemplo completo de tipo anotación.....	26
6.- META-ANOTACIONES PREDEFINIDAS.....	27
6.1.- @Target.....	28
6.2.- @Retention.....	29
6.3.- @Inherited.....	31
6.4.- @Documented.....	32
6.5.- @Repeatable.....	34
7.- ANOTACIONES PREDEFINIDAS.....	35
7.1.- @Deprecated.....	35
7.2.- @Override.....	36
7.3.- @SupressWarnings.....	37
7.4.- @SafeVarargs.....	39
7.5.- @FunctionalInterface.....	40
8.- ANÁLISIS Y DISEÑO DE TIPOS ANOTACIÓN.....	41
8.1.- Ventajas de las anotaciones.....	41
8.2.- Inconvenientes de las anotaciones.....	41
8.3.- Cuándo usar anotaciones (y cuándo no).....	42
8.4.- Análisis de tipos anotación.....	43
8.5.- Diseño de tipos anotación.....	44
8.6.- Conclusiones.....	47
9.- PROCESAMIENTO DE ANOTACIONES.....	48
9.1.- Conceptos básicos.....	48

9.2.- Procesamiento según la política de retención.....	49
9.2.1.- Política de retención SOURCE.....	50
9.2.2.- Política de retención CLASS.....	50
9.2.3.- Política de retención RUNTIME.....	51
9.3.- Métodos de procesamiento de anotaciones.....	51
9.3.1.- Compilador Java + Procesadores de anotaciones.....	52
9.3.2.- Herramientas de manipulación de ficheros de clase.....	53
9.3.3.- Código Java usando reflexión.....	55
10.- PROCESAMIENTO J2SE 1.5.....	58
10.1.- Introducción.....	58
10.2.- Cómo funciona el procesamiento J2SE 1.5.....	59
10.3.- Componentes del procesamiento J2SE 1.5.....	62
10.3.1.- Procesadores de anotaciones J2SE 1.5.....	62
10.3.1.1.- Procesadores de anotaciones.....	62
10.3.1.2.- Factorías de procesadores de anotaciones.....	63
10.3.1.3.- Descubrimiento de declaraciones a procesar.....	66
10.3.1.4.- Opciones para los procesadores desde línea de comandos.....	70
10.3.1.5.- Distribución de procesadores J2SE 1.5 en ficheros .jar.....	75
10.3.2.- Mirror API.....	77
10.3.2.1.- Paquete com.sun.mirror.apt.....	78
10.3.2.2.- Paquete com.sun.mirror.declaration.....	81
10.3.2.3.- Paquete com.sun.mirror.type.....	82
10.3.2.4.- Paquete com.sun.mirror.util.....	83
10.3.2.5.- Declaraciones vs Tipos.....	85
10.3.2.6.- Mirrors: A través del espejo.....	88
10.3.2.7.- Filtrado de declaraciones.....	93
10.3.2.8.- Usos del patrón Visitor.....	94
10.3.3.- apt (annotation processing tool).....	100
10.3.3.1.- Descripción de apt.....	100
10.3.3.2.- Funcionamiento de apt.....	101
10.3.3.3.- Opciones de línea de comandos de apt.....	107
10.3.3.4.- Invocación de apt.....	109
10.3.3.5.- Fallos conocidos y limitaciones de apt.....	118
10.4.- Procesamiento J2SE 1.5: Paso a paso.....	119
10.4.1.- Análisis y Diseño del tipo anotación.....	119
10.4.2.- Implementación del Procesador de anotaciones.....	120
10.4.3.- Anotación de elementos de código.....	125
10.4.4.- Invocación de apt y obtención de resultados.....	126
10.5.- Ejemplos: Procesadores J2SE 1.5.....	128
10.5.1.- Navegación básica: @ImprimirMensaje.....	128
10.5.2.- Navegación avanzada: @Nota.....	129
10.5.3.- Validación básica: @Contexto.....	136
10.5.4.- Validación avanzada: @JavaBean.....	139
10.5.5.- Generación de código básica: @Proxy.....	143
10.5.6.- Generación de código avanzada: @AutoTest.....	149
10.6.- Migración de procesamiento J2SE 1.5 a JSR 269.....	157
10.7.- aptIn16 (apt en Java SE 6+).....	159
11.- NOVEDADES EN JAVA SE 6.....	160
11.1.- Novedades generales en Java SE 6.....	160
11.2.- Common Annotations (JSR 250).....	161
11.3.- Java Compiler API (JSR 199).....	161
11.4.- Compiler Tree API.....	161
11.5.- Pluggable Annotation Processing API (JSR 269).....	162

12.- COMMON ANNOTATIONS (JSR 250).....	163
12.1.- Common Annotations 1.0 (Java SE 6, Java EE 5).....	164
12.1.1.- Common Annotations 1.0 para Java SE 6.....	165
12.1.1.1.- @Generated.....	165
12.1.1.2.- @Resource.....	166
12.1.1.3.- @Resources.....	169
12.1.1.4.- @PostConstruct.....	170
12.1.1.5.- @PreDestroy.....	171
12.1.2.- Common Annotations 1.0 para Java EE 5.....	172
12.1.2.1.- @DeclareRoles.....	172
12.1.2.2.- @RolesAllowed.....	173
12.1.2.3.- @PermitAll.....	174
12.1.2.4.- @DenyAll.....	175
12.1.2.4.- @RunAs.....	176
12.2.- Common Annotations 1.1 (Java EE 6).....	177
12.2.1.- @ManagedBean.....	177
12.2.2.- @DataSourceDefinition.....	178
12.2.3.- @DataSourceDefinitions.....	180
12.3.- Common Annotations 1.2 (Java EE 7).....	181
12.3.1.- @Priority.....	181
13.- JAVA COMPILER API (JSR 199).....	182
13.1.- Introducción.....	182
13.2.- Paquete javax.tools.....	183
14.- COMPILER TREE API.....	185
14.1.- Introducción.....	185
14.2.- Paquete com.sun.source.tree.....	186
14.3.- Paquete com.sun.source.util.....	189
15.- PROCESAMIENTO JSR 269 (JAVA SE 6+).	190
15.1.- Introducción.....	190
15.2.- Cómo funciona el procesamiento JSR 269.....	191
15.3.- Componentes del procesamiento JSR 269.....	196
15.3.1.- Procesadores de anotaciones JSR 269.....	197
15.3.1.1.- Procesadores de anotaciones.....	197
15.3.1.2.- Descubrimiento de elementos a procesar.....	202
15.3.1.3.- Descubrimiento de elementos usando Compiler Tree API.....	205
15.3.1.4.- Resumen de resultados del proceso de descubrimiento.....	209
15.3.1.5.- Opciones para los procesadores desde línea de comandos.....	210
15.3.1.6.- Distribución de procesadores JSR 269 en ficheros .jar.....	213
15.3.2.- API de procesamiento: javax.annotation.processing.....	215
15.3.3.- Language Model API: javax.lang.model.....	217
15.3.3.1.- Paquete javax.lang.model.element.....	218
15.3.3.2.- Paquete javax.lang.model.type.....	220
15.3.3.3.- Paquete javax.lang.model.util.....	221
15.3.3.4.- Elementos vs Tipos.....	225
15.3.3.5.- Mirrors: A través del espejo.....	228
15.3.3.6.- Uso del patrón Visitor.....	236
15.3.4.- javac.....	253
15.3.4.1.- Funcionamiento de javac.....	253
15.3.4.2.- Opciones de línea de comandos de javac.....	259
15.3.4.3.- Invocación de javac.....	261
15.4.- Procesamiento JSR 269: Paso a paso.....	280
15.4.1.- Análisis y Diseño del tipo anotación.....	280
15.4.2.- Implementación del Procesador de anotaciones.....	281

15.4.3.- Anotación de elementos de código.....	285
15.4.4.- Invocación del procesamiento y obtención de resultados.....	286
15.5.- Ejemplos: Procesadores JSR 269.....	288
15.5.1.- Navegación básica: @ImprimirMensaje.....	288
15.5.2.- Navegación avanzada: @Nota.....	289
15.5.3.- Validación básica: @Contexto.....	298
15.5.4.- Validación avanzada: @JavaBean.....	301
15.5.5.- Generación de código básica: @Proxy.....	306
15.5.6.- Generación de código avanzada: @AutoTest.....	314
15.6.- Depuración de procesadores de anotaciones.....	328
16.- TRANSFORMACIÓN DE CÓDIGO FUENTE.....	329
16.1.- Introducción.....	329
16.1.- Javaparser.....	329
16.2.- Spoon.....	330
16.3.- Project Lombok.....	331
17.- NOVEDADES EN JAVA SE 7.....	332
17.1.- Novedades generales en Java SE 7.....	332
17.2.- Novedades Java SE 7 en APIs de procesamiento.....	333
17.2.1.- Novedades Java SE 7: API de reflexión.....	333
17.2.2.- Novedades Java SE 7: Language Model API.....	333
17.2.3.- Novedades Java SE 7: Compiler Tree API.....	338
18.- NOVEDADES EN JAVA SE 8.....	339
18.1.- Novedades generales en Java SE 8.....	339
18.2.- Novedades Java SE 8 en APIs de procesamiento.....	340
18.2.1.- Novedades Java SE 8: API de reflexión.....	340
18.2.2.- Novedades Java SE 8: Language Model API.....	345
18.2.3.- Novedades Java SE 8: Compiler Tree API.....	348
18.3.- Novedades del procesamiento en Java SE 8.....	350
18.3.1.- Eliminación definitiva de apt (JSR 269 / JEP 117).....	350
18.3.2.- Anotaciones repetibles (JSR 269 MR 2 / JEP 120).....	352
18.3.3.- Anotaciones sobre tipos y proc. variables locales (JSR 308 / JEP 104).....	352
19.- ANOTACIONES SOBRE TIPOS (JSR 308).....	353
19.1.- Un sistema de tipos mejorado.....	353
19.2.- Nuevos @Target: TYPE_PARAMETER y TYPE_USE.....	355
19.3.- Anotaciones sobre parámetros de tipo.....	355
19.4.- Anotaciones sobre usos de tipos.....	356
19.4.1.- Concepto de “uso de tipo”	356
19.4.2.- Ejemplos: Anotaciones sobre usos de tipos correctas.....	357
19.4.3- Ejemplos: Anotaciones sobre usos de tipos incorrectas	358
19.4.4- Anotaciones multi-target con múltiples elementos afectados.....	359
19.4.5- Sintaxis de tipo receptor explícito.....	360
19.5.- Carencias del procesamiento hasta Java SE 8.....	363
19.5.1.- Información incompleta en el fichero de clase.....	364
19.5.2.- Procesamiento de anotaciones sobre variables locales.....	365
19.5.3.- Procesamiento de anotaciones sobre parámetros de tipo.....	365
20.- PROCESAMIENTO EN JAVA SE 8.....	366
20.1.- Introducción.....	366
20.2.- Problema: Descubrimiento de anotaciones sobre tipos.....	366
20.3.- Uso de las diferentes APIs para el procesamiento.....	368
20.4.- Ejemplos de código.....	369
21.- CHECKER FRAMEWORK.....	370
21.1.- Introducción.....	370
21.2.- Funcionamiento.....	371

21.3.- Plug-in para Eclipse.....	372
22.- NOVEDADES EN JAVA SE 9.....	375
22.1.- Novedades generales en Java SE 9.....	375
22.2.- Novedades Java SE 9 en APIs de procesamiento.....	376
22.2.1.- Novedades Java SE 9: API de reflexión.....	376
22.2.2.- Novedades Java SE 9: Language Model API.....	376
22.2.3.- Novedades Java SE 9: Compiler Tree API.....	379
22.2.4.- Novedades Java SE 9: API de procesamiento JSR 269.....	380
22.3.- Novedades del procesamiento en Java SE 9.....	381
22.3.1.- Pipeline de anotaciones 2.0 en el compilador (JEP 217).....	381
23.- CONCLUSIONES.....	382
24.- BIBLIOGRAFÍA.....	383
24.1.- Libros.....	383
24.2.- Páginas web.....	384
24.2.1.- Páginas generales de referencia.....	384
24.2.2.- Recursos de desarrollo.....	384
24.2.3.- Documentación técnica básica.....	385
24.2.4.- Documentación técnica adicional.....	386
24.2.5.- Artículos.....	387
24.2.6.- Tutoriales.....	387
25.- HISTORIAL DE CAMBIOS.....	388

1.- ACERCA DEL MANUAL.

1.1.- Estructuración del contenido.

El presente manual ha tratado de ser exhaustivo en cuanto a la cobertura de los temas más importantes acerca de las anotaciones y su procesamiento, acompañando su contenido de muchos ejemplos, algunos desarrollados en profundidad con fragmentos de código, lo cual redunda en una gran cantidad de secciones y contenidos.

Este manual puede ser leído de forma secuencial, pero se han estructurado sus contenidos de forma muy pormenorizada para facilitar su uso como documento rápido de referencia.

Se ha planteado la evolución de los contenidos de forma cronológica, empezando por el origen y la evolución de la plataforma Java y la introducción de las anotaciones en J2SE 1.5.

A partir de ahí, se presentan los conceptos básicos referidos a las anotaciones y su procesamiento y se pasa a explicar todos los detalles sobre las anotaciones y los tipos anotación.

El siguiente bloque se compone de la referencia rápida de las meta-anotaciones y anotaciones predefinidas en la plataforma Java SE, de obligado conocimiento para todo desarrollador.

A continuación, con los conceptos sobre anotaciones y tipos anotación bien asentados, pasamos a ver algunos conceptos sobre las ventajas e inconvenientes de las anotaciones, cuando utilizarlas (y cuando no) y las directrices y principios fundamentales a la hora de llevar a cabo el análisis y diseño de nuestros propios tipos anotación.

Seguidamente, dedicamos un apartado al procesamiento de anotaciones en J2SE 1.5, actualmente en desuso y a extinguir, pero que se incluye en este manual a efectos didácticos.

Tras J2SE 1.5, seguimos con un apartado dedicado a las novedades introducidas en Java SE 6: una guía de referencia de las Commons Annotations, la Java Compiler API, la Compiler Tree API y un apartado monográfico sobre el procesamiento de anotaciones introducido en Java SE 6 usando la nueva JSR 269 (Pluggable Annotation Processing API).

A continuación del bloque dedicado a las tecnologías introducidas en Java SE 6, pasamos a examinar las novedades en Java SE 7 y Java SE 8, especialmente las Anotaciones sobre Tipos (JSR 308) y los cambios introducidos en el procesamiento de anotaciones Java SE 8. Para finalizar, analizamos las novedades de Java SE 9, la última versión de la plataforma Java hasta la fecha .

1.2.- Itinerarios de lectura.

Introducción al procesamiento de anotaciones

Capítulos 2 a 9.

50 páginas aproximadamente.

Procesamiento de anotaciones clásico

Capítulo 10.

100 páginas aproximadamente.

Procesamiento de anotaciones moderno

Capítulos 11 a 17.

200 páginas aproximadamente.

Procesamiento de anotaciones avanzado (incluyendo anotaciones sobre tipos)

Capítulos 18 a 22.

100 páginas aproximadamente.

1.3.- Anexos de código fuente.

Junto con el manual se ha incluido como anexo el código fuente de los ejemplos completos que ilustran en detalle el procesamiento de anotaciones. También se incluyen los códigos de algunas de las pruebas detalladas de código que se explican a lo largo del manual para aclarar conceptos, de cara a que el lector pueda verlos y ejecutarlos por sí mismo.

Recomendamos al lector que examine detenidamente estos anexos, ya que el código fuente de los mismos está escrito desde un punto de vista didáctico, por lo que se han incluido completos comentarios de código que explican detalladamente todo el proceso, incluyendo características peculiares de las APIs de procesamiento, buenas prácticas, consejos, trucos, etcétera.

Los ficheros anexos de código fuente incluidos pertenecen a un *workspace* del entorno de desarrollo [Eclipse](#), concretamente [Eclipse para desarrolladores Java EE versión Oxygen 1a](#), por lo que es muy fácil ponerlos en funcionamiento rápidamente, ya que, una vez descargado, sólo hay que seleccionar el directorio de dicho *workspace* al arrancar dicho entorno y configurar los entornos de Java.

Todos los proyectos de ejemplo incluidos utilizan [Maven](#) para la gestión de dependencias y del proceso de construcción (*build*), por lo que incluyen en su raíz el correspondiente “pom.xml”. El uso de Maven permite además usar los plugins de Maven que integran el procesamiento de anotaciones en la consola de Eclipse para ver la salida de los procesadores, tal y como está explicado en los apartados “Invocación de apt” (J2SE 1.5) e “Invocación de javac” (Java SE 6+).

Finalmente, en la raíz de los proyectos también se incluyen diferentes ficheros .bat para invocar los ejemplos directamente sobre el compilador Java de Oracle (y no el de Eclipse), puesto que el comportamiento y la salida de ambos compiladores puede variar ligeramente en algunos casos.

2.- ORIGEN DE LAS ANOTACIONES.

2.1.- Creación de la plataforma Java.

El proyecto de creación del lenguaje Java se inició en Sun Microsystems en el año 1991 por parte de un equipo de ingenieros liderado por James Gosling. Diseñado originalmente para la TV interactiva, el lenguaje se demostró muy avanzado para la tecnología de su época, y pronto se vio que podría servir a propósitos más ambiciosos.

Inicialmente llamado Oak (roble) en referencia a un roble que había fuera del despacho de James Gosling, al encontrarse dicha marca ya registrada, finalmente acabó llamándose Java, por el café de Java que los creadores del lenguaje consumían en grandes cantidades.

Desde el principio, se pretendió que Java implementara una máquina virtual y que el lenguaje tuviera una sintaxis muy parecida a C++, de forma que muchos programadores pudieran familiarizarse de forma cómoda y rápida con Java. De hecho, se establecieron cinco **principios básicos para el diseño del lenguaje Java**:

- 1) Que fuera "sencillo, orientado a objetos y familiar".
- 2) Que fuera "robusto y seguro".
- 3) Que fuera "de arquitectura neutral y portable".
- 4) Que se ejecutara con un "alto rendimiento".
- 5) Que fuera "interpretado, multihebra, y dinámico".

Sun Microsystems publicó la primera implementación de Java en 1995, con el lema de "Write once, run anywhere" ("Escríbelo una vez, ejecútalo en cualquier parte") y que prometía tiempos de ejecución sin apenas ningún coste en las plataformas de desarrollo más populares.

Dadas las excelentes capacidades de seguridad, los principales navegadores web pronto incorporaron la capacidad de ejecutar *applets* Java, lo cual hizo al lenguaje enormemente popular y conocido. En 1999 llegó "Java 2", que introdujo sus plataformas de desarrollo clásicas:

- Java 2 Standard Edition (J2SE) para el desarrollo general de aplicaciones.
- Java 2 Enterprise Edition (J2EE) para el desarrollo para entornos empresariales.
- Java 2 Micro Edition (J2ME) para dispositivos limitados, como tablets, PDAs o móviles.

El lenguaje Java ha sufrido muchos cambios desde su primera versión, así como en su biblioteca de clases estándar, que ha pasado de tener pocos cientos de clases a varios miles. Para controlar su evolución, se creó el Java Community Process (JCP), un organismo formado por particulares y empresas que se encargan de proponer especificaciones de cambio o nuevas características para el lenguaje en forma de Java Specification Requests (JSRs).

Cuando, tras varios procesos de revisión, finalmente se aprueba una JSR pasa a formar parte del "canon oficial" de Java. Y, de hecho, si alguna JSR afecta al propio lenguaje, sus cambios se incorporan a la especificación oficial: la Java Language Specification (JLS). Siguiendo esta metodología colaborativa, desde J2SE 1.4, en el año 2002, el JCP ha venido dirigiendo la evolución de Java, incorporando muchas novedades importantes.

2.2.- Evolución de la plataforma Java hasta J2SE 1.5.

Como hemos introducido en el apartado anterior, en la década transcurrida desde el año 1995 hasta la aparición de J2SE 1.5 en el año 2004, Java ha demostrado una enorme evolución.

La evolución del lenguaje Java ha sido un proceso muy interesante gracias a la creación del Java Community Process (JCP), un organismo formado por expertos seleccionados, ya sean particulares y/o empresas, que ha sido el encargado de recibir propuestas de especificaciones de cambios o nuevas características en forma de Java Specification Requests (JSRs).

Este subapartado es un repaso muy somero a la evolución de la plataforma de desarrollo Java (ya no sólo del lenguaje), para poner al lector en situación acerca de los cambios y novedades que sufrió Java desde su creación y que han provocado que sea tal y como es. Llegaremos hasta J2SE 1.5 para introducir al lector en el contexto en el que se disponían de las facilidades dadas por la herramienta **apt** y el procesamiento de anotaciones J2SE 1.5.

NOTA: En algunos apartados del manual pueden incluirse puntualmente algunas referencias a características de versiones posteriores de la plataforma Java, especialmente de Java SE 6 así como Java SE 8, que han sido las versiones que más novedades han introducido en cuanto al procesamiento de anotaciones. Dicha información se incluye por completitud, pero la intención del manual es que el lector vaya conociendo las nuevas características del lenguaje en un orden lo más similar posible al orden cronológico en que fueron presentadas. Para más información sobre las novedades de cada una de las versiones posteriores a J2SE 1.5 de la plataforma Java, este manual dedica un apartado “Novedades en Java SE X” para cada una de las mismas.

Versiones de la plataforma Java hasta J2SE 1.5			
Versión	Nombre / Codename	Fecha Salida	Novedades más importantes
JDK (Java Development Kit) 1.0	Java 1	23/ENE/1996	Versión pública inicial desde el inicio del desarrollo en el año 1995.
JDK (Java Development Kit) 1.1	Java 1.1	19/FEB/1997	Clases internas , Reflexión , JavaBeans , JDBC , RMI , Compilador JIT en Windows.
J2SE (Java 2 Standard Edition) 1.2	Playground	8/DIC/1998	Introducción de las ediciones de plataformas de desarrollo: J2SE , J2EE y J2ME . Java plug-ins/applets , Colecciones , Swing , strictfp , Compilador JIT para la JVM.
J2SE (Java 2 Standard Edition) 1.3	Kestrel	8/MAY/2000	HotSpot JVM, JNDI , JPDA (depuración), RMI compat. CORBA , Reflexión mejorada.
J2SE (Java 2 Standard Edition) 1.4 [JSR 59]	Merlin	6/FEB/2002	assert , encadenamiento de excepciones , Java Web Start , NIO , Logging API , Image IO , JAXP (parser XML, procesador XSLT).
J2SE 1.5 / J2SE 5.0 [JSR 176]	Tiger	30/SEP/2002	Mejoras: Genericidad , Autoboxing , enum , “ for mejorado ”, @Anotaciones , varargs , import static , java.util.concurrent , correcciones al Modelo de Memoria Java para evitar problemas con la concurrencia, seguridad, y facilitar las optimizaciones.

Como podemos ver en la tabla, J2SE 1.4 ya fue dirigido por la especificación [JSR 59](#) del Java Community Process (JCP). A partir de entonces, la plataforma Java se iría desarrollando en base a las propuestas de la comunidad de desarrolladores Java. Las propuestas aprobadas se van desarrollando en especificaciones Java Specification Request (JSR) y finalmente se agrupan bajo un “JSR paraguas” que aglomere los JSR que se publican en cada versión de la plataforma.

2.3.- Anotaciones en J2SE 1.5.

Desde que nació Java, y antes de que tuviera soporte de anotaciones, algunos opinan que espiritualmente ya tenía algunos mecanismos de meta-information, un tanto *sui generis* eso sí, que transmitían metadatos de elementos del código, como eran el modificador `transient` (que indica que una variable de instancia no ha de ser serializada) o el tag Javadoc `@deprecated` (que indica que un método está “desfasado” y será eliminado en el futuro, por lo que es mejor evitar usarlo), y que luego devendría en la anotación `@Deprecated`, que sirve para “anotar” exactamente lo mismo. De hecho, las anotaciones complementan las etiquetas JavaDoc y, si lo que se pretende es generar documentación, aún se recomienda utilizar JavaDoc; si no se pretende crear documentación, deberían utilizarse anotaciones.

Las anotaciones se introdujeron en Java con en **J2SE 1.5**, habiendo sido previamente formalizadas en la especificación [JSR 175](#) (A Metadata Facility for the Java™ Programming Language), que introdujo los tipos anotación predefinidos para las anotaciones en `java.lang` (`@Deprecated`, `@Override`, `@SuppressWarnings`) y las meta-anotaciones predefinidas en `java.lang.annotation` (`@Target`, `@Retention`, `@Inherited` y `@Documented`).

No obstante, aunque el diseño de las anotaciones estaba bien ideado, su soporte no se realizó con un criterio tan afortunado. Y es que el procesado de las anotaciones, que de forma natural correspondía al compilador Java, fue desplazado a una herramienta externa llamada `apt` (annotation processing tool). Rápidamente se vio que el **apt estaba mal diseñado**, además de que la API de la que hacía uso, **la Mirror API (paquete com.sun.mirror)**, **no soportaba el procesamiento de anotaciones sobre variables locales**.

Además, en la propia especificación del lenguaje **no se permitía herencia entre tipos anotación**, dificultando con ello la creación de mejores y más elegantes diseños, así como limitando los beneficios de la reutilización de código en lo que respecta a los tipos anotación. Esta limitación, a diferencia de las otras, está justificada como medio para evitar problemas de compilación del lenguaje y no está prevista que se implemente en ningún momento en el futuro.

Para acometer estos problemas, [Mathias Ricken](#), de la Rice University de Houston (Texas, EEUU), creó versiones modificadas/hacks de `apt` y `javac`. No se recomienda su uso, puesto que no fueron integradas como parte de la plataforma estándar de Java. No obstante, podrían ser útiles en casos de extrema necesidad y servir de caso de estudio por motivos académicos:

LAPT-javac: Local Variable Annotation Enabled javac [<http://www.cs.rice.edu/~mgricken/research/laptjavac/>] Versión modificada de `apt` que permite procesar las anotaciones sobre variables locales. No obstante, estas no son visibles para la Mirror API ni la API de reflexión, siendo inservibles para su procesamiento mediante procesadores de anotación al nivel de retención de código fuente.

xajavac: Extended Annotation Enabled javac [<http://www.cs.rice.edu/~mgricken/research/xajavac/>] Versión modificada de `javac` que permite herencia de tipos anotación. Esta vez sí se han introducido los cambios oportunos en la API de reflexión para acceder a los datos de los nuevos tipos de anotación heredados. No obstante, se utilizan clases especiales *ad hoc* y ubicadas en un paquete no oficial, cuya documentación se pueden consultar en: <http://www.cs.rice.edu/~mgricken/research/xajavac/download/subannot.javadoc/>

Muchos de los problemas de uso de `apt` se solventarán en Java SE 6 con la introducción de la especificación [JSR 269](#) (Pluggable Annotation Processing API). Otros, como el soporte de procesamiento de variables locales, tendrán que esperar hasta Java SE 8 con la introducción de [JSR 308](#) (Annotation on Java Types).

3.- CONCEPTOS BÁSICOS.

3.1.- Concepto de anotación.

Se define el concepto de “**anotación**”, dentro del ámbito del lenguaje Java, como aquel **mecanismo sintáctico que permite añadir al código fuente información adicional**, denominada habitualmente como “meta datos”.

Las anotaciones son expresiones del lenguaje Java con una sintaxis propia y que se aplican como si fueran un tipo de modificador más sobre otros elementos del lenguaje (tales como paquetes, clases, campos, métodos, parámetros o variables locales). Este proceso de aplicar anotaciones sobre un elemento de código se conoce como “anotar” o, en algunas raras ocasiones, “decorar”.

Aunque entraremos en detalles en todo lo referente a las anotaciones en el siguiente apartado, la **sintaxis básica de una anotación** tiene la siguiente forma:

Carácter @ + Nombre de la anotación + (pares elemento = valor)

Ejemplo: Anotación con varios elementos

```
// la siguiente anotación utiliza elementos de diferentes tipos, incluyendo un enumerado
@AnotacionPrueba(cadena="abc", numero=12, decimal=3.3f, color=EnumColor.VERDE)
public ObjetoEjemplo() { ... }
```

Después de anotar el código, la Máquina Virtual Java u otras herramientas podrán examinar la meta-information ofrecida por las anotaciones para determinar cómo interaccionar con los elementos de programa anotados y modelar su comportamiento.

En el ejemplo anterior de `@SupressWarnings` es el compilador el que actúa de cierta manera al detectar una anotación que está anotando en este caso la definición de un método. La anotación `@SupressWarnings`, que es una anotación predefinida estándar en Java SE, da instrucciones al compilador de que no emita warnings de cierto tipo (en ese caso los de tipo “unused”) sobre los elementos de códigos que anota. En este caso, el compilador no emitirá avisos sobre variables no utilizadas referidas al método, sus argumentos o las variables locales que se definan dentro de su cuerpo.

Las anotaciones no afectan directamente a la semántica del código pero, como hemos visto, sí pueden afectar a cómo dicho código es procesado por otras herramientas, por lo que sí pueden afectar indirectamente a la semántica concreta de ejecución de una aplicación.

No se preocupe si no entiende aún cómo encajan todos los detalles del funcionamiento a más bajo nivel de las anotaciones. A lo largo del presente manual se explicará cada componente y cada paso del proceso en profundidad. Lo importante ahora es que se familiarice con los conceptos más básicos e importantes: el hecho de que las anotaciones son un elemento sintáctico más del lenguaje que permite adjuntar información adicional a otros elementos del código fuente para que dicha información pueda ser procesada posteriormente.

3.2.- Procesamiento de anotaciones.

Una vez el código fuente está anotado, en una etapa posterior del proceso de desarrollo, **las anotaciones serán reconocidas y procesadas de diversas formas** a partir de sus propiedades concretas, ya sea en tiempo de compilación o de ejecución. Este proceso se conoce como “**procesamiento de anotaciones**”.

Las anotaciones pueden ser leídas de ficheros de código fuente, de los ficheros de clase o en tiempo de ejecución a través de los mecanismos de reflexión de la API de Java SE.

En función de cómo estén diseñadas, las anotaciones nos permitirán realizar desde acciones sencillas y directas, como simplemente informar a los programadores de ciertas propiedades del código, hasta otras acciones más complejas, como configuración o generación de nuevos ficheros de código fuente o ficheros de recursos. Así pues, las anotaciones pueden ser:

- 1) Anotaciones pasivas: son puramente informativas y su procesamiento no genera nada nuevo.
- 2) Anotaciones activas: su procesamiento genera nuevos elementos (código fuente y/o ficheros).

El hecho de que el procesar una anotación pueda dar lugar a nuevo código fuente que haya a su vez que compilar, y así sucesivamente, convierte el procesamiento de anotaciones en un proceso iterativo, compuesto de lo que se conoce como “rondas” (“rounds” en el inglés original) de compilación. El compilador irá ejecutando una ronda de compilación tras otra hasta que no se genere ningún elemento nuevo a compilar.

Al igual que con las anotaciones, no se preocupe si aún no le quedan claros los detalles de cómo funciona el proceso del procesamiento de anotaciones. Más adelante en el manual hay varios apartados específicos dedicados a este proceso.

Conviene tener en cuenta que existen varios apartados dedicados al procesamiento de anotaciones porque la forma de llevarlo a cabo cambió entre J2SE 1.5 y Java SE 6, y a partir de entonces ha ido siendo mejorado progresivamente en las versiones posteriores de la plataforma.

El presente manual sólo incluye la forma de realizar el procesamiento para J2SE 1.5, considerado desfasado y a extinguir, por completitud, dado su carácter didáctico. Actualmente se recomienda utilizar el método moderno de procesamiento de anotaciones instroducido en Java SE 6 llamado JSR 269 (Pluggable Annotation Processing API). Encontrará todos los detalles sobre este tipo de procesamiento en el apartado “**PROCESAMIENTO JSR 269 (JAVA SE 6+)**”.

3.3.- Usos habituales de las anotaciones.

Las anotaciones sirven para **modelar propiedades de los elementos del código fuente**, por ejemplo:

- Anotar restricciones de uso de un elemento: @Deprecated, @Override
- Anotar la naturaleza o el tipo de un elemento: @Entity, @TestCase, @WebService
- Anotar el comportamiento de un elemento: @Statefull, @Transaction
- Anotar la forma de procesar un elemento: @Column, @XmlElement

Otros usos más prácticos de las anotaciones suelen ser aquellos que facilitan, mediante la anotación del código y el consecuente enriquecimiento de la información asociada al mismo, una extracción o procesado de dicha información como plantillas de generación de código fuente y/o ficheros de recursos de diverso tipo, entre los que se incluyen:

- Configuración de aplicaciones (sustituyendo al XML).
- Activación de comportamientos de cierto tipo para agilizar el desarrollo (prototipado).
- Programación declarativa y programación orientada a aspectos (AOP).
- Configuración de clases y métodos para APIs de testing (JUnit es una de las más conocidas).
- Logging (registro) de eventos de la aplicación para su depuración.
- Documentación (XDoclet) / Meta-Información para IDE's (Eclipse, NetBeans) o el compilador.
- Mecanismos de marcado de generación de código automático (como clientes de servicios web).
- Serialización automática de información a disco o cualquier otro mecanismo de persistencia.

Las anotaciones constituyen actualmente una herramienta de capital importancia para los desarrolladores más avanzados, especialmente los de librerías, que nutren sus APIs con un buen número de anotaciones para facilitar el proceso de desarrollo a sus usuarios.

Ejemplos de librerías importantes (o incluso de plataformas de desarrollo oficiales) que usan profusamente las anotaciones son APIs tan conocidas y populares como Spring, Hibernate, o como la propia especificación Java Enterprise Edition (Java EE).

4.- ANOTACIONES.

4.1.- Sintaxis.

Toda anotación siempre se escribe **empezando por el carácter arroba @**, que es el carácter distintivo que identifica a una anotación de código en Java. A continuación, **le sigue el nombre del tipo anotación** propiamente dicho y **la lista de valores de los elementos**. Dichos valores, como veremos más adelante, deberán ser valores constantes definidos en tiempo de compilación.

Una anotación puede escribir con o sin elementos. Si la anotación tiene elementos, se escribe una lista de múltiples pares clave-valor de tantos elementos como para los que se quiera establecer un valor. Si una anotación no tiene elementos la lista de valores será simplemente (), aunque lo más habitual en ese caso es ni siquiera escribir dicho par de paréntesis.

Así pues, en función de si una anotación tiene elementos o no, podemos encontrarnos con diversas formas sintácticas correctas a la hora de usar una anotación en el lenguaje Java:

```
@Anotacion  
@Anotacion(ValorElementoUnico) → sintaxis especial para anotaciones con un único elemento  
@Anotacion(NombreElem1 = ValorElem1, NombreElem2 = ValorElem2, ...)
```

A continuación, pasaremos a explicar más detalladamente y con diversos ejemplos las peculiaridades de cada una de las formas sintácticas con que pueden escribirse las anotaciones.

4.2.- Variantes sintácticas.

4.2.1.- Anotación normal.

Sintaxis: `@ + Nombre de la anotación + (pares elemento = valor)`

Esta es la construcción sintáctica más general y completa para escribir anotaciones. Las variantes sintácticas simplificadas que vamos a ver después siempre pueden escribirse siguiendo esta sintaxis, ya que lo único que hacen es omitir o simplificar la sintaxis de lista de pares elemento = valor, al tener uno o ningún elemento.

Ejemplo 1: Anotación con un par de elementos sobre una clase

```
// anotación sencilla con un par de elementos informativos de la versión del método  
@AnotacionVersion(version="1.1", fecha="31/08/2013") // no se usa Date para la fecha  
public class ClaseEjemplo() { ... }
```

Ejemplo 2: Anotación con elementos de varios tipos en un constructor

```
// la siguiente anotación utiliza elementos de diferentes tipos, incluyendo un enumerado  
@AnotacionElementos(nombre="Miguel", numero=99, decimal=3.3f, color=EnumColor.VERDE)  
public ObjetoPrueba() { ... }
```

4.2.2.- Anotación marcadora (Marker Annotation).

Sintaxis: @ + Nombre de la anotación

Los tipos anotación que no definen ningún elemento propio se conocen como anotaciones “marcadoras”. El término se refiere a que suelen utilizarse para señalar / apuntar / marcar algún tipo de propiedad, característica o naturaleza específica que no requiere de ningún tipo de información más allá de la presencia de la propia anotación.

Ejemplo: Anotación marcadora (sin elementos) sobre un método

```
@Override // esta anotación indica que este método está redefiniendo algún otro
public String toString() {

    // llamamos a la implementación de la superclase, en este caso java.lang.Object
    return super.toString();
}
```

4.2.3.- Anotación sencilla (Single Element Annotation).

Sintaxis: @ + Nombre de la anotación + (valor)

Las anotaciones “sencillas”, o “de elemento único”, son aquellas que sólo definen un único elemento. La especificación del lenguaje Java define una sintaxis especial muy elegante para ellas que estas podrán utilizar siempre y cuando su único elemento se llame “**value**”. Si no fuera así, tendría que utilizarse forzosamente la sintaxis completa de las anotaciones normales.

Ejemplo 1: Anotación que elimina los warnings sobre un método

```
@SuppressWarnings("unused") //también podría escribirse @SuppressWarnings(value="unused")
// hace que el compilador no arroje warnings de tipo "variable no utilizada" en el método
public static void main(String[] args) { ... }
```

Ejemplo 2: Anotación que elimina el warning de variable no utilizada

```
@SuppressWarnings("unused") //también podría escribirse @SuppressWarnings(value="unused")
// la anotación hace que el compilador no arroje warnings de tipo "variable no utilizada"
int variableNoUtilizada = 0;
```

Ejemplo 3: Anotación que elimina warnings sobre elementos “desfasados”

```
// instanciamos una fecha con un constructor desfasado
// y añadimos una anotación para que el compilador
// no se queje acerca de este hecho arrojando un warning
@SuppressWarnings("deprecation") // podría escribirse
@SuppressWarnings(value="deprecation")
Date fecha = new Date(113, 07, 31); // crea la fecha 31/08/2013
```

4.3.- Elementos de código anotables.

Es posible “anotar” todos los distintos elementos del lenguaje de programación Java como: paquetes, clases, constructores, métodos, variables, etcétera.

Para anotar un elemento de código del programa, simplemente hay que escribir la anotación previamente al elemento de código siguiendo cualquiera de las variantes sintácticas correctas. A ser posible, es mejor siempre ubicarlas en una línea anterior y no en la misma línea, salvo en las anotaciones sobre parámetros de métodos, como veremos en los ejemplos.

Hay que seguir un cierto procedimiento especial únicamente en caso de que queramos anotar paquetes: deberemos crear un fichero **package-info.java** en el directorio raíz del paquete con la declaración de paquete, descripción JavaDoc y colocar en dicho fichero, sobre la declaración propiamente dicha del paquete, las anotaciones correspondientes.

Incluso es posible anotar anotaciones con otras anotaciones. Estas “**anotaciones sobre anotaciones**”, se conocen con el nombre de **meta-anotaciones** y, de hecho, el lenguaje Java incluye algunas meta-anotaciones predefinidas muy importantes a la hora de definir tipos anotación (@Target, @Retention, @Inherited y @Documented).

Ejemplo 1: Anotación sobre un paquete

Según la especificación del lenguaje, la anotación de un paquete sólo puede hacerse desde un fichero especial llamado **package-info.java** (siempre tiene que tener este nombre, sea cual sea el nombre del paquete) situado en la raíz del directorio correspondiente al paquete en cuestión.

Fichero package-info.java (situado en el directorio del paquete es.uma.lcc.anotaciones)

```
/**  
 * Este paquete contiene todo el código de ejemplo del PFC sobre anotaciones Java.  
 */  
@AnotacionEjemploElementosMultiples(  
    activa=true,  
    nombre="Miguel",  
    numero=666,  
    decimal=0.666f,  
    color=AnotacionEjemploElementosMultiples.EnumeradoColor.AZUL)  
package es.uma.lcc.anotaciones;
```

Ejemplo 2: Anotación sobre una clase

```
@AnotacionEjemploSobreClase(autor="MIRM", version="1.1")  
public class ClaseEjemplo() { ... }
```

Ejemplo 3: Anotación sobre un constructor

```
@AnotacionEjemploSobreConstructor(pre="init", post="ready")  
public ClaseEjemplo() { ... }
```

Ejemplo 4: Anotación sobre un método

```
@AnotacionEjemploSobreMetodo(opcion="escribir", fichero="GeneracionCodigo.java")  
public void MetodoEjemplo() { ... }
```

Ejemplo 5: Anotación sobre un parámetro de un método

En el caso de que las anotaciones vayan sobre el parámetro de un método la sintaxis puede ser algo más confusa y complicada de leer a simple vista, como en el siguiente caso:

```
public void imprimirInfo(@SuppressWarnings("rawtypes") Class clase, File fichero) { ... }
```

No obstante, si se desea mantener una buena legibilidad es muy sencillo reescribir el método escalonando su definición y sus parámetros en diferentes líneas de la siguiente forma:

```
public void imprimirInfo(
    @SuppressWarnings("rawtypes") Class clase,
    File fichero) {
    ...
}
```

Ejemplo 6: Anotación sobre una variable

```
@AnotacionEjemploSobreVariable(serializar=false, thread-safe=true)
int variable = 0;
```

Ejemplo 7: Anotación sobre una anotación (meta-anotación)

```
@Documented // esta meta-anotación indica que @AnotacionEjemplo se documenta en JavaDoc
public @interface AnotacionEjemplo { ... }
```

4.4.- Ubicación de las anotaciones.

A la hora de ubicar una anotación, hay que tener en cuenta que el compilador trata las anotaciones como si fueran un modificador más (como `public`, `static` o `final`). Por tanto, **las anotaciones pueden ubicarse allí donde puedan ubicarse los modificadores**. El siguiente fragmento de código compila sin problemas:

```
public static @AnotacionEjemploElementosMultiples(
    activa=true, nombre="Miguel", numero=666, decimal=0.666f,
    color=EnumeradoColor.AZUL) void main(String[] args) { ... }
```

No obstante, esta construcción, que entremezcla la anotación con los modificadores de la firma del método, es claramente confusa y poco legible, por lo cual se desaconseja utilizarla. Y es que, aunque no es en absoluto estrictamente obligatorio, ya que el compilador Java es flexible, **por convención, cualquier anotación siempre se incluye antes de cualquier otro modificador estándar**. Siguiendo esta convención escribiríamos:

```
@AnotacionEjemploElementosMultiples(
    activa=true,
    nombre="Miguel",
    numero=666,
    decimal=0.666f,
    color=EnumeradoColor.AZUL)
public static void main(String[] args) { ... }
```

Esta segunda construcción sintáctica es mucho más legible y es por ello que es la preferida según la convención habitual recomendada por los diseñadores del lenguaje.

4.5.- Número de anotaciones.

Los elementos de código pueden anotarse con un **número ilimitado** de anotaciones. Normalmente, **cuando hay que etiquetar un mismo elemento de código con varias anotaciones, se “apilan” unas encima de otras**, precediendo todas ellas al resto de los modificadores, como comentábamos en el apartado dedicado a la ubicación de las anotaciones.

No obstante, es importante aclarar que, **aunque se pueden apilar anotaciones sin límite** sobre un elemento de código anotable, **no se puede repetir la misma anotación para un mismo elemento de código**.

Ejemplo 1: Anotaciones múltiples sobre un método

```
@Entity  
@Table(name="TABLA_CLASE")  
public class Clase { ... }
```

Ejemplo 2: Anotaciones múltiples repetidas → error de compilación

```
@Entity  
@Table(name="TABLA_CLASE")  
@Table(name="OTRA_TABLA") // anotación Table repetida -> error de compilación  
public class Clase { ... }
```

No se pueden repetir anotaciones para un mismo elemento. Si, como en el ejemplo, se quisiera especificar más de una tabla, se podría diseñar la anotación para reconocer múltiples valores dentro del valor de un String y separados por un carácter separador o haciendo que el elemento name del tipo anotación fuera un array String[], en cuyo caso se podría establecer su valor con un inicializador de array de la siguiente manera: `@Table(name={"TABLA_CLASE", "OTRA_TABLA"})`.

IMPORTANTE: En Java SE 8 se ha solucionado esta restricción implementando una solución similar a la propuesta aquí para poder permitir apilar anotaciones del mismo tipo sobre una misma declaración. La idea es, dejando intacta la definición del tipo anotación original, poder definir un nuevo tipo anotación que haga de “contenedor”. Para más detalles, ver el apartado siguiente sobre “Anotaciones repetibles (Java SE 8)”.

Ejemplo 3: Anotaciones múltiples sobre la definición de un tipo anotación

Como para definir un tipo anotación y establecer sus propiedades es normal que haya que emplear una o varias meta-anotaciones, es muy común ver apiladas anotaciones encima de las definiciones de tipos anotación.

```
@Retention(RetentionPolicy.RUNTIME) // anotación en 1 línea -> lo más habitual  
@Target({ElementType.PACKAGE, ElementType.TYPE, ElementType.CONSTRUCTOR, // anotación  
         ElementType.METHOD, ElementType.FIELD, ElementType.LOCAL_VARIABLE}) // en 2 líneas  
@Documented @Inherited // 2 anotaciones en 1 única línea; se permite una gran  
flexibilidad  
public @interface AnotacionEjemploElementosMultiples { ... }
```

4.6.- Anotaciones repetibles (Java SE 8+).

En Java SE 8 se ha añadido al lenguaje una nueva característica: [anotaciones repetibles](#). Es decir: la posibilidad de poder diseñar tipos anotación repetibles que permitan realizar múltiples anotaciones sobre el mismo elemento de código.

Antes de Java SE 8, una sintaxis como la siguiente provocaba un error de compilación:

```
@AnotacionRepetible(nombre="Miguel", edad=33)
@AnotacionRepetible(nombre="Ángel", edad=30) → error de compilación
public void inicializar() { ... }
```

Con Java SE 8 se introduce una nueva meta-anotación estándar llamada **@Repeatable** (en inglés: “repetible”) que nos permitirá definir otro tipo anotación que hará de “contenedor” para las múltiples instancias del “tipo anotación contenido” original. Esto se hace así para poder mantener la retro-compatibilidad del lenguaje con código fuente anterior a Java SE 8, ya que el “tipo anotación contenido” original será una anotación normal.

La definición oficial de la nueva meta-anotación **@Repeatable** es la siguiente:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Repeatable {
    Class<? extends Annotation> value();
}
```

Así pues, si queremos que nuestra `@AnotacionRepetible` sea en efecto repetible, habremos de anotarla con la meta-anotación `@Repeatable` de la siguiente manera:

```
@Repeatable(AnotacionesRepetibles.class)
// AnotacionesRepetibles es el "tipo anotación contenedor"
@Retention(RetentionPolicy.RUNTIME)
public @interface AnotacionRepetible {

    String nombre() default "Miguel"; // null daría error de compilación
    int edad() default 18;
}
```

Nótese que la meta-anotación `@Repeatable` tiene como único elemento `value`, que debe ser una clase, que será, como adelantábamos, el “tipo anotación contenedor” que Java utilizará para guardar las múltiples apariciones del “tipo anotación contenido” original.

En nuestro ejemplo, el “tipo anotación contenido” es `@AnotacionRepetible` y el “tipo anotación contenedor” sería `@AnotacionesRepetibles` (la convención es utilizar los términos plurales del nombre del tipo anotación contenido original), y que estaría definido como sigue:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface AnotacionesRepetibles {
    AnotacionRepetible[] value() default { }; // por defecto: array vacío
}
```

IMPORTANTE: Hay que tener en cuenta que al consultar las anotaciones asociadas a una declaración a través de la API de reflexión del lenguaje Java (siempre que la política de retención del tipo anotación sea `RetentionPolicy.RUNTIME`), usando los métodos de la interfaz `java.lang.reflect.AnnotatedElement`, los métodos que originalmente sólo devolvían una anotación, como `<T extends Annotation> T getAnnotation(Class<T> claseAnotacion)`, seguirán devolviendo una anotación. Si se quiere tener acceso a las múltiples ocurrencias del “tipo anotación contenido”, como valor de `claseAnotacion` se le puede pasar la clase del “tipo anotacion contenedor”. De esta forma devolverá una única instancia, pero del contenedor, con lo cual, a través del array `value` podremos acceder a todas las ocurrencias del “tipo anotacion contenido”. En nuestro caso, por ejemplo, si tomamos una referencia al método anotado (`inicializar`) y consultamos su anotación mediante el siguiente código:

```
Method metodoInicializar = getMethod("inicializar", (Class<?>[])null);
Annotation anotacion = metodoInicializar.getAnnotation(AnotacionRepetible.class);
System.out.println("anotacion = " + anotacion);
```

Obtenemos la siguiente salida:

```
anotacion = @anotaciones.ejemplos.definicion.AnotacionRepetible(edad=33, nombre=Miguel)
```

Así pues, como hemos adelantado, sólo obtenemos una de las varias anotaciones `@AnotacionRepetible` con las que hemos anotado la declaración del método. No obstante, si indicamos como clase la del “tipo anotación contenedor” (`AnotacionesRepetibles.class`), obtendremos todas las anotaciones repetidas a través del valor de su elemento `value`:

```
anotacion =
@anotaciones.ejemplos.definicion.AnotacionesRepetibles(value=[@anotaciones.ejemplos.definicion.AnotacionRepetible(edad=33, nombre=Miguel),
@anotaciones.ejemplos.definicion.AnotacionRepetible(edad=30, nombre=Ángel)])
```

También podemos utilizar los métodos que devuelven una colección de anotaciones:

```
Annotation anotaciones[] = metodoInicializar.getAnnotations(); // sin especificar clase
System.out.println("anotaciones = " + Arrays.toString(anotaciones));
```

Que devuelve exactamente lo mismo que el método anterior:

```
anotaciones =
[@anotaciones.ejemplos.definicion.AnotacionesRepetibles(value=[@anotaciones.ejemplos.definicion.AnotacionRepetible(edad=33, nombre=Miguel),
@anotaciones.ejemplos.definicion.AnotacionRepetible(edad=30, nombre=Ángel)])]
```

5.- TIPOS ANOTACIÓN.

Toda anotación usada en el lenguaje Java tiene previamente definido su correspondiente “tipo anotación”, que no es más que la definición de sus respectivas propiedades y elementos que la componen de cara a que el compilador pueda identificarla.

Possiblemente, uno de los principales añadidos que introdujo la especificación **JSR 175** en el lenguaje fue la posibilidad de que **los programadores pueden definir sus propios tipos anotación personalizados**.

5.1.- Sintaxis.

La **definición de los tipos anotación** se asemeja sintácticamente a la definición de una interfaz Java, pero con varias particularidades, restricciones y construcciones especiales:

- Los tipos anotación se definen como una interfaz con una @ antes de interface (@interface).
- Los tipos anotación no son heredables (lo cual es un inconveniente de cara al diseño).
- Los elementos de las anotaciones se definen mediante métodos con una sintaxis restringida: no pueden declarar excepciones usando throws ni tomar parámetros de entrada.
- Los elementos pueden tener valores por defecto usando default.
- Los elementos sólo pueden ser de ciertos tipos concretos, según la especificación del lenguaje, y que pasaremos a detallar en el siguiente apartado.

Ejemplo: Definición de un tipo anotación con varios elem. de diverso tipo

```
public @interface TipoAnotacionEjemplo {  
  
    // métodos de definición de elementos de diversos tipos de retorno  
    // los métodos no podrán tener argumentos ni declarar excepciones con "throws",  
    // aunque sí podrán declarar valores por defecto con "default"  
    boolean activa() default true;  
    String nombre() default "porDefecto";  
    int numero() default 32;  
    float decimal() default 3.14159f;  
}
```

5.2.- Tipos permitidos para los elementos.

Los tipos de retorno de los valores de los métodos que definen los elementos de un tipo anotación no pueden ser de cualquier tipo, sino que **están restringidos a los siguientes**:

- Tipos primitivos (boolean, byte, int, char, short, long, float y double).
- String.
- Enumerados.
- Class, tal cual o con parámetros de tipo (Class<T>, Class<? extends Collection>).
- Otros Tipos Anotación previamente definidos.
- Un array de cualquiera de los tipos anteriores.

Ejemplo: Definición de un tipo anotación con los distintos tipos permitidos

```
public @interface TipoAnotacionEjemploTiposPermitidos {  
  
    // elementos tipo primitivo: boolean, byte, int, char, short, long, float y double  
    boolean elementoBoolean() default false;  
    byte elementoByte() default 0b00100001; // 00100001 = 33 en formato binario  
    int elementoInt() default Integer.MAX_VALUE;  
    char elementoChar() default '\uffff';  
    short elementoShort() default Short.MAX_VALUE;  
    long elementoLong() default Long.MAX_VALUE;  
    float elementoFloat() default Float.MAX_VALUE;  
    double elementoDouble() default Double.MAX_VALUE;  
  
    // resto de tipos de retorno permitidos para los elementos  
    String elementoString() default "Por Defecto";  
    EnumeradoColor elementoEnumerado() default EnumeradoColor.ROJO;  
    Class elementoClass() default ArrayList.class;  
    Class<Hashtable> elementoParametrizado() default Hashtable.class;  
    Class<? extends List> elementoClassParametrizadoComodin() default ArrayList.class;  
    AnotacionEjemplo elementoOtraAnotacion() default @AnotacionEjemplo(value=true);  
    Class<? extends Collection>[] elementoClassParametrizadoComodinArray()  
        default { HashSet.class, LinkedList.class };  
}
```

5.3.- Valores permitidos y por defecto en elementos.

Los valores por defecto serán asignados a los elementos en caso de que no se les asigne un valor de forma explícita. No obstante, si no se define un valor por defecto para un elemento y dicho valor no se especifica en la anotación propiamente dicha, el compilador dará un error exigiendo que se asigne un valor, ya que **no se permitirá que ningún elemento se quede sin valor.**

Otro detalle importante es que **los valores de los elementos de una anotación nunca podrán ser null, aunque su tipo declarado lo soporte** (como en el caso de la clase String). Ya habíamos mencionado que los valores de los elementos debían ser siempre constantes con un valor claro en tiempo de compilación (nada de llamadas a métodos), pero la restricción en este caso va incluso un poco más allá al no poder usar tampoco valores nulos.

5.4.- Meta-anotaciones sobre los tipos anotación.

La definición de un tipo anotación habitualmente está, a su vez, anotada. Un tipo anotación que puede anotar a otras anotaciones se conoce con el nombre de tipo meta-anotación o simplemente meta-anotación.

Las meta-anotaciones, que describimos en profundidad en el apartado correspondiente de este manual, serán necesarias precisamente a la hora de escribir la definición de un tipo anotación, donde se emplearán para especificar propiedades concretas acerca de la naturaleza de los nuevos tipos anotación que se quieran definir.

Aunque hasta ahora no entendemos para que sirven las meta-anotaciones del siguiente ejemplo, como las fundamentales @Retention y @Target, poco más adelante en este manual explicaremos en detalle las características de las diferentes meta-anotaciones predefinidas dadas por el lenguaje Java para permitirnos definir nuestros propios tipos anotación.

Ejemplo: Definición de un tipo anotación con diversas meta-anotaciones

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.PACKAGE, ElementType.TYPE, ElementType.CONSTRUCTOR,
         ElementType.METHOD, ElementType.FIELD, ElementType.LOCAL_VARIABLE})
@Documented
@Inherited
public @interface AnotacionEjemploConMetaAnotaciones { ... }
```

5.5.- Cuadro de referencia de un tipo anotación.

Como veremos en el apartado dedicado a las Meta-Anotaciones predefinidas, los tipos anotación se definirán fundamentalmente en base a dos propiedades básicas: la dada por su “target” (u “objetivo”) y la dada por su “retention” (o “política de retención”), pero hay otras.

En el resto de este documento, cada vez que queramos describir un tipo anotación, la acompañaremos de un cuadro resumen de referencia rápida con sus principales propiedades.

Además, en el cuadro de referencia se incluirán otras propiedades importantes de los tipos anotación y que resulte práctico de conocer rápidamente para el desarrollador, como su firma o la definición de sus elementos. La estructura de los cuadros resumen es la siguiente:

@Anotacion	
Descripción	Descripción breve que ayude a identificar la funcionalidad fundamental del tipo anotación.
Clase	Nombre completamente cualificado de la clase que modela el tipo anotación.
Target	Lista de valores TARGET sobre los que se puede aplicar el tipo anotación.
Retention	Política de retención del tipo anotación.
Otros	Lista de otras meta-anotaciones adicionales aplicadas sobre el tipo anotación.
Desde	Versión de la arquitectura Java a partir de la cual el tipo anotación está disponible.
Signatura	Signatura ejemplo del tipo anotación, donde pueden verse sus elementos con un valor concreto.
Elementos	Lista de elementos propios del tipo anotación y una explicación breve de sus valores válidos.

Por ejemplo, para el tipo anotación **@Generated**, una de las Commons Annotations, el cuadro resumen de referencia quedaría de la siguiente manera:

@Generated	
Descripción	Anotación que permite marcar código generado por herramientas de generación de código.
Clase	<code>javax.annotation.Generated</code>
Target	<code>PACKAGE, TYPE, ANNOTATION_TYPE, METHOD, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, PARAMETER</code>
Retention	<code>SOURCE</code>
Otros	<code>@Documented</code>
Desde	Java SE 6, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@Generated(value = "es.uma.lcc.PFCAnotaciones.ClaseGeneradoraCodigo", comments = "Este código está generado automáticamente", date = "2014-04-23T15:30:56.235-0100") // fecha formato ISO 8601</code>
Elementos	<code>value</code> = Clase generadora del código. Nombre completamente cualificado, incluyendo paquete. <code>comments</code> = Comentario libre que puede meter la clase generadora del código a voluntad. <code>date</code> = Fecha del momento en que el código fue generado.

Estos cuadros facilitarán el uso de este manual como instrumento de referencia rápida a la hora de consultar información importante de una forma cómoda sobre cualquiera de las anotaciones o meta-anotaciones cubiertas en el mismo con sólo localizarlas en el índice y acudiendo a su apartado propio, donde encontrará con el cuadro correspondiente a la misma.

5.6.- Ejemplo completo de tipo anotación.

El siguiente ejemplo completo refleja todos los aspectos sintácticos de la definición de un tipo anotación profusamente comentados para que el lector pueda tener una visión completa.

En el ejemplo se usan prácticamente todas las posibilidades y variantes sintácticas existentes a la hora de escribir la definición de un tipo anotación: meta-anotaciones varias, elementos con diversos tipos de retorno (incluyendo un tipo enumerado) y con sus respectivos valores por defecto.

Lo único que no aparece en el ejemplo son un tipo array o un tipo anotación como tipo de retorno de uno de sus elementos, o parámetros de tipo con comodines, cosa que, aunque no es lo más común, está permitido por las reglas de definición de los tipos anotación, como hemos visto en el apartado “Tipos permitidos para los elementos”.

Fichero AnotacionEjemploElementosMultiples.java

```
package es.uma.lcc.anotaciones;

import java.lang.annotation.*;

/**
 * Ejemplo de definición de una anotación de múltiples elementos.
 */
@Retention(RetentionPolicy.RUNTIME) // anotación visible en tiempo de ejecución
// definimos múltiples targets para la anotación (si no definimos un Target,
// por defecto la anotación se puede aplicar a cualquier elemento del código)
@Target({ElementType.PACKAGE, ElementType.TYPE, ElementType.CONSTRUCTOR,
         ElementType.METHOD, ElementType.FIELD, ElementType.LOCAL_VARIABLE})
@Documented // incluye la anotación en el proceso de generación de documentación de Javadoc
@Inherited // anotación heredable por las subclases de la clase anotada
public interface AnotacionEjemploElementosMultiples {

    // definimos un enumerado local para demostrar que los tipos enumerados
    // también pueden ser tipos válidos
    public enum EnumeradoColor { ROJO, VERDE, AZUL }

    // como podemos observar, para definir una anotación hemos de escribir una
    // sintaxis parecida a la declaración de una interfaz, pero con una @ delante

    // cada método que declaremos de la interfaz representará un elemento de la anotación,
    // por lo que los métodos no podrán tener argumentos ni declarar excepciones con "throws",
    // aunque sí podrán declarar valores por defecto con "default" para los casos en los que
    // los programadores que hagan uso de la anotación no especifiquen un valor concreto
    boolean activa() default true;
    String nombre() default "porDefecto"; // null daría error de compilación
    int numero() default 32;
    float decimal() default 3.14159f;
    EnumeradoColor color() default EnumeradoColor.AZUL;

    // con la definición realizada para la anotación, ya podríamos anotar un elemento de código
    // por ejemplo de las siguientes formas:
    // @AnotacionEjemploElementosMultiples -> todos los elementos a sus valores por defecto
    // @AnotacionEjemploElementosMultiples(activa=true, nombre="Miguel", numero=99,
    // decimal=3.3f, color=EnumeradoColor.VERDE)
}
```

6.- META-ANOTACIONES PREDEFINIDAS.

La especificación [JSR 175](#) (A Metadata Facility for the Java™ Programming Language), a partir de J2SE 1.5, determinó la forma en que se definen y usan los tipos anotación.

Precisamente para que los programadores pudieran definirse sus propios tipos anotación, fue necesario que **se incluyeran “de serie” varios tipos meta-anotación predefinidos** que serían necesarios **para modelar las propiedades de los tipos anotación**.

Los siguientes **tipos meta-anotación predefinidos** son imprescindibles para definir las propiedades fundamentales de los tipos anotación del lenguaje Java:

Tipos Meta-Anotación predefinidos (paquete <code>java.lang.annotation</code>)	
Anotación	Descripción
<code>@Target (ElementType[] value)</code>	Indica los elementos de código sobre los que se aplica el tipo anotación. Sus posibles valores vienen dados por el enumerado <code>ElementType</code> : <code>ANNOTATION_TYPE</code> (una anotación aplicable a un tipo anotación es una meta-anotación), <code>CONSTRUCTOR</code> , <code>FIELD</code> (campos, dinámicos y estáticos, y constantes de enums), <code>LOCAL_VARIABLE</code> , <code>METHOD</code> , <code>PACKAGE</code> , <code>PARAMETER</code> , <code>TYPE</code> (clases, interfaces, anotaciones y declaraciones de enums). Por defecto, si no se indica <code>@Target</code> , la anotación se podrá aplicar sobre todos los elementos de código.
<code>@Retention (RetentionPolicy[] value)</code>	Indica la política de retención del tipo anotación, según los valores de la enumeración <code>RetentionPolicy</code> : <code>CLASS</code> (valor por defecto: la anotación se guarda en el fichero de clase, para que lo pueda leer el compilador, pero no es accesible en tiempo de ejecución vía reflexión), <code>RUNTIME</code> (la anotación se guarda en el fichero de clase y estará disponible también en tiempo de ejecución vía reflexión) y <code>SOURCE</code> (disponible únicamente a nivel de código fuente y no en posteriores etapas; tipo anotación descartada por el compilador).
<code>@Inherited</code>	Indica que las anotaciones del tipo anotación anotado sobre una determinada clase C, afectan también las subclases de C.
<code>@Documented</code>	Indica que un tipo anotación debe ser incluido en la generación de documentación de JavaDoc y similares, haciendo aparecer por tanto la anotación documentada como parte de la API pública, además de aparecer documentada en el JavaDoc generado en cada uno de los elementos anotados por dicho tipo anotación.
<code>@Repeatable</code>	Indica que un tipo anotación es repetible, es decir: que pueden aplicarse múltiples anotaciones de dicho tipo anotación sobre un mismo elemento del código fuente. Introducida en Java SE 8.

6.1.- @Target

@Target	
Descripción	Meta-anotación que indica los tipos de elementos de programa anotables por un tipo anotación.
Clase	<code>java.lang.annotation.Target</code>
Target	<code>ANNOTATION_TYPE</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 5, JSR 275 (A Metadata Facility for the Java™ Programming Language)
Signatura	<code>@Target({ElementType.TYPE, ElementType.CONSTRUCTOR})</code>
Elementos	<code>value = ElementType[]</code> . Array de ElementType, uno por cada tipo de elemento anotable.

La meta-anotación `@Target` sirve para indicar los elementos de código fuente Java que pueden servir como objetivo del tipo anotación (“target” significa “objetivo” en inglés).

Por defecto, si no se anota la definición de un tipo anotación con `@Target`, el tipo de anotación será aplicable a cualquier elemento de código fuente que pueda ser anotado.

Los **valores válidos** para el elemento value de la meta-anotación `@Target` vienen dados por la clase enumerada `java.lang.annotation.ElementType` y son los siguientes:

Valores de <code>java.lang.annotation.ElementType</code> permitidos como valores válidos de <code>@Target</code>	
Tipo de elemento	Descripción
<code>ANNOTATION_TYPE</code>	Tipo anotación aplicable a otros tipos anotación. Este valor define a una meta-anotación. Por este motivo, se da la circunstancia curiosa de que <code>@Target</code> anota su propia definición de tipo con el valor <code>@Target(value=ANNOTATION_TYPE)</code> sobre sí misma.
<code>CONSTRUCTOR</code>	Tipo anotación aplicable a constructores.
<code>FIELD</code>	Tipo anotación aplicable a campos: dinámicos (variables de instancia) y estáticos (variables de clase), y a constantes de tipos enumerados.
<code>LOCAL_VARIABLE</code>	Tipo anotación aplicable a variables locales.
<code>METHOD</code>	Tipo anotación aplicable a métodos.
<code>PACKAGE</code>	Tipo anotación aplicable a paquetes.
<code>PARAMETER</code>	Tipo anotación aplicable a parámetros de métodos o constructores.
<code>TYPE</code>	Tipo anotación aplicable a lo que la especificación del lenguaje Java llama “tipo”, un concepto amplio que incluye clases, interfaces, tipos anotación, así como declaraciones de tipos enumerados.
<code>TYPE_PARAMETER</code> (Java SE 8+)	Tipo anotación aplicable a parámetros de tipo.
<code>TYPE_USE</code> (Java SE 8+)	Tipo anotación aplicable a “usos” de tipos. Para entender el concepto de “uso” de un tipo, véase el apartado “Anotaciones sobre tipos”.

Ejemplo 1: @Target anotando un tipo anot. que será aplicado sólo a métodos

```
@Target(ElementType.METHOD)
public @interface TipoAnotacion { ... }
```

Ejemplo 2: @Target con múltiples valores

```
@Target({ElementType.PACKAGE, ElementType.TYPE, ElementType.CONSTRUCTOR,
          ElementType.FIELD, ElementType.METHOD, ElementType.LOCAL_VARIABLE})
public @interface TipoAnotacion { ... }
```

Ejemplo 3: Definición del propio tipo anotación @Target

Lo siguiente es la definición oficial del tipo anotación `@Target`, donde puede verse cómo dicha definición está anotada por varias meta-anotaciones, entre ellas la propia `@Target` con el valor `ANNOTATION_TYPE`, que es la que convierte a `@Target` en una meta-anotación.

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    ElementType[] value();
}
```

6.2.- @Retention

@Retention	
Descripción	Indica la política de retención del tipo anotación a través del ciclo de desarrollo.
Clase	<code>java.lang.annotation.Retention</code>
Target	<code>ANNOTATION_TYPE</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 5, JSR 275 (A Metadata Facility for the Java™ Programming Language)
Signatura	<code>@Retention(RetentionPolicy.RUNTIME)</code>
Elementos	<code>value</code> = Una de las políticas de retención dadas por el tipo enumerado <code>RetentionPolicy</code> .

La meta-anotación `@Retention` indica hasta qué punto debe retenerse (“retention” en inglés significa “retención”), es decir, mantenerse por parte de la plataforma Java, la información acerca de las anotaciones del tipo anotación anotado con ella.

La meta-anotación `@Retention` sólo tiene efecto sobre la declaración de tipos anotación. No tiene efecto alguno sobre instancias del tipo anotación anotado que se utilicen como elementos en otros tipos anotación.

El valor de `@Retention` es lo que la especificación del lenguaje denomina “**política de retención**” y sus valores válidos vienen dados por el tipo enumerado `RetentionPolicy`.

Por defecto, si no se anota la definición de un tipo anotación con esta meta-anotación, la política de anotación aplicada sobre dicho tipo anotación será `CLASS`. No obstante, es raro el caso en que un tipo anotación no es anotado con `@Retention`. Los valores válidos son los siguientes:

Valores de <code>java.lang.annotation.RetentionPolicy</code> permitidos como valores válidos de <code>@Retention</code>	
Política de retención	Descripción
<code>SOURCE</code>	La anotación sólo estará disponible a nivel del código fuente. El compilador no la incluirá en los ficheros de clase generados.
<code>CLASS</code>	Valor por defecto. La anotación estará disponible a nivel de código, y se guardará en los ficheros de clase generados por el compilador, pero no será accesible en tiempo de ejecución a través de la API de reflexión.
<code>RUNTIME</code>	La anotación se guarda en los ficheros de clase generados, y también estará disponible en tiempo de ejecución a través de la API de reflexión.

Las **etapas del ciclo de desarrollo que atraviesa el código Java**, desde que se escribe hasta que se ejecuta en una implementación concreta de la Máquina Virtual Java, y con las que está estrechamente relacionada el tipo anotación `@Retention`, son, en orden cronológico:

- 1) Compilación del código fuente (SOURCE).**
- 2) Generación del fichero de clase (CLASS).**
- 3) Ejecución en la Máquina Virtual Java (RUNTIME).**

La visibilidad de las anotaciones en cada una de estas etapas dependerá de la política de retención de su tipo anotación correspondiente, y es que **el valor de la política de retención define cuál es la última etapa en la cual es visible el tipo anotación**.

Es decir, que la visibilidad viene dada por un esquema incremental siguiendo el propio orden cronológico de las etapas del ciclo de desarrollo Java, según el cual si el tipo anotación se define con una retención de una determinada etapa, será visible para todas las anteriores:

Visibilidad de los tipos anotación según la Política de Retención dada por <code>@Retention</code>			
Visibilidad \ Retención	Política de Retención <code>SOURCE</code>	Política de Retención <code>CLASS</code>	Política de Retención <code>RUNTIME</code>
Visible en <code>SOURCE</code>	SÍ	SÍ	SÍ
Visible en <code>CLASS</code>	NO	SÍ	SÍ
Visible en <code>RUNTIME</code>	NO	NO	SÍ

Por este motivo, un tipo anotación al que se le anote con una política de retención `RUNTIME` (última etapa del ciclo de desarrollo en orden cronológico), estará visible siempre. O los tipos anotación con política de retención `CLASS` estarán visibles en `SOURCE` y `CLASS`, pero no en `RUNTIME`, que es la siguiente fase.

Finalmente, los tipos anotación que se definan con la política de retención `SOURCE`, al ser esta la primera fase del ciclo de desarrollo, sólo estarán visibles en dicha etapa y en ninguna más de las posteriores. Es por ello que las anotaciones de tipos de anotación con retención `SOURCE` sólo están disponibles en el código fuente original de las aplicaciones y se pierde (no guardándose o no “reteniéndose”) cuando el compilador genera los ficheros de clase.

Ejemplo: `@Retention` con una política de retención `RUNTIME` (la más habitual)

```
@Retention(RetentionPolicy.RUNTIME)
public @interface AnotacionEjemplo { ... }
```

6.3.- @Inherited

@Inherited	
Descripción	Indica que las anotaciones del tipo anotación sobre una clase afectan también a sus subclases.
Clase	<code>java.lang.annotation.Inherited</code>
Target	<code>ANNOTATION_TYPE</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 5, JSR 275 (A Metadata Facility for the Java™ Programming Language)
Signatura	<code>@Inherited</code>
Elementos	N/D (anotación marcadora)

La meta-anotación `@Inherited` sirve para indicar que un tipo anotación es heredable, es decir: que las anotaciones de este tipo anotación que se efectúen sobre la declaración de una clase, son heredadas automáticamente por todas sus subclases.

Por defecto, si no se anota la definición de un tipo anotación con `@Inherited`, el tipo anotación no será heredable a las subclases de las clases anotadas con dicho tipo anotación.

Si una meta-anotación no es `@Inherited` para determinar si una clase está anotada con un tipo anotación concreto vía reflexión simplemente se busca en los meta-datos de la propia clase. Sin embargo, si la meta-anotación es `@Inherited`, de decir, que es un “tipo anotación heredable”, al consultar vía reflexión si dicho “tipo anotación heredable” está presente en una determinada clase C, se irá consultando desde la definición en sí de la clase C hacia arriba, de superclase en superclase, hasta que se encuentre el “tipo anotación heredable” buscado en alguna de las superclases, en cuyo caso se considerará que el tipo anotación está, por herencia, presente en la clase C, o hasta que se alcance la cima de la jerarquía de clases (la clase Object) sin encontrarlo, en cuyo caso se considerará que el tipo anotación no está presente en la clase C.

NOTA 1: `@Inherited` no tiene efecto alguno si el “tipo anotación heredable” se usa para usar un elemento de código distinto a una clase. Es decir, que sólo se heredan las anotaciones de un “tipo anotación heredable” que estén anotando la declaración de una clase.

NOTA 2: `@Inherited` sólo provoca que se hereden anotaciones desde las superclases de una clase y no desde las interfaces o superinterfaces implementadas por una clase. Esto es así porque, como ya hemos visto, la búsqueda de los tipos de anotación heredables únicamente se realiza en la jerarquía de herencia de clases, y no se mira ninguna otra cosa.

Ejemplo: @Inherited anotando un tipo anot. y convirtiéndolo en heredable

```
@Inherited  
public @interface TipoAnotacionHeredable { ... }
```

6.4.- @Documented

@Documented	
Descripción	Indica que un tipo anotación y sus anotaciones deben ser documentadas en JavaDoc.
Clase	<code>java.lang.annotation.Documented</code>
Target	<code>ANNOTATION_TYPE</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 5, JSR 275 (A Metadata Facility for the Java™ Programming Language)
Signatura	<code>@Documented</code>
Elementos	N/D (anotación marcadora)

La meta-anotación `@Documented` sirve para indicar que un tipo anotación es un “tipo anotación documentable”, y debe ser incluido en los procesos de generación de documentación de herramientas como JavaDoc y similares, haciendo aparecer por tanto el tipo anotación anotado con `@Documented` como parte de la API pública, además de aparecer documentado en la documentación JavaDoc cada uno de los elementos anotados por dicho tipo anotación.

IMPORTANTE: Para que `@Documented` tenga [en J2SE 1.5](#) el efecto deseado, debido a cómo se procesaban las anotaciones en J2SE 1.5, es posible que sea necesario que el tipo anotación que anotemos con `@Documented` deba tener una **retención RUNTIME**. Este requerimiento se ha probado que no es necesario a partir de Java SE 7 e incluso es muy probable ni siquiera lo sea en Java SE 6 (versión en la que se introdujo la nueva implementación del procesamiento de anotaciones).

Por defecto, si no se anota la definición de un tipo anotación con `@Documented`, el tipo anotación no se mostrará como parte de la API pública en JavaDoc, ni aparecerán sus valores en la documentación JavaDoc generada para los elementos anotados por dicho tipo anotación.

`@Documented` debería usarse para dar visibilidad a los tipos anotación que los desarrolladores quieran mostrar como parte de su API pública. De esta forma, estos “tipos anotación públicos” serán visibles para los clientes de la API pública y podrán ser utilizados por ellos, así como podrán visualizar a lo largo del JavaDoc de la API pública cuando cualquier elemento de código esté notado por anotaciones de dichos “tipos anotación públicos”.

`@Documented` no debería usarse en los tipos anotación que los desarrolladores usen únicamente de modo interno para llevar a cabo la implementación de sus aplicaciones, y cuyo uso no esté previsto para los clientes externos de dicha API pública. Si estos “tipos anotación internos”, “tipos anotación privados” o “tipos anotación de implementación” (como se quiera denominarlos) no se marcan con `@Documented`, los clientes de la API pública ni siquiera sabrán de su existencia, al igual que con los elementos de visibilidad `private` del lenguaje.

Ejemplo: @Documented anotando un tipo anot., haciéndolo documentable

Lo siguiente es una definición de un tipo anotación cualquiera al cual hemos anotado con **@Documented**, ya que queremos que la definición de dicho tipo anotación esté disponible para los clientes de nuestra API pública, así como que se muestren las anotaciones de dicho tipo anotación que se hagan en todos los demás elementos de código Java que sean documentados.

```
@Documented  
public @interface AnotacionElementosMultiples { ... }
```

Al haber sido marcado como **@Documented**, el tipo anotación se muestra en JavaDoc, lo siguiente es un extracto de su página JavaDoc generada, con su definición y sus elementos:

es.uma.lcc.PFCAnotaciones.ejemplos.anotaciones.definicion

Annotation Type AnotacionElementosMultiples

```
@Retention(value=RUNTIME)  
@Target(value={PACKAGE,TYPE,CONSTRUCTOR,METHOD,FIELD,LOCAL_VARIABLE})  
@Documented  
@Inherited  
public @interface AnotacionElementosMultiples
```

Este fichero de código sirve de ejemplo para demostrar la definición de una anotación de múltiples elementos.

Optional Element Summary

Optional Elements	Modifier and Type	Optional Element and Description
	boolean	activa
	AnotacionElementosMultiples.EnumeradoColor color	
	float	decimal
	java.lang.String	nombre
	int	numero

Además de la definición del “tipo anotación documentable”, en el JavaDoc generado se mostrarán las anotaciones de dicho tipo anotación sobre los elementos de la API pública. A continuación, podemos ver cómo un método público de una clase de ejemplo que se ha marcado con el “tipo anotación documentable” **@AnotacionElementosMultiples** es mostrado en JavaDoc:

metodoAnotadoPorTipoAnotacionDocumented

```
@AnotacionElementosMultiples(nombre="Una anotacion de un tipo anotacion con @Documented se muestra en javadoc.")  
public void metodoAnotadoPorTipoAnotacionDocumented(java.lang.String mensaje)
```

6.5.- @Repeatable

@Repeatable	
Descripción	Indica que el tipo anotación es un tipo anotación repetible para un mismo elemento de código.
Clase	<code>java.lang.annotation.Repeatable</code>
Target	<code>ANNOTATION_TYPE</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 8
Signatura	<code>@Repeatable</code>
Elementos	<code>value = Class</code> . Clase contenedora donde acumular las repeticiones del tipo anotado.

La meta-anotación `@Repeatable` sirve para indicar que un tipo anotación es repetible, es decir, un tipo anotación que puede anotar múltiples veces el mismo elemento de código. Las [anotaciones repetibles](#) son una nueva característica introducida en el lenguaje en Java SE 8. Para más detalles, véase el apartado “[Anotaciones repetibles \(Java SE 8\)](#)” de este manual.

Para que un tipo-anotación sea repetible, es preciso definir otro tipo anotación que defina un array de la anotación repetible, una especie de “tipo anotación contenedor”. En dicho “tipo anotación contenedor” será donde el entorno de ejecución Java guardará las referencias a las múltiples anotaciones sobre un elemento de código del tipo anotación repetible.

Ejemplo: Definiendo un tipo anotación repetible con @Repeatable

Para definir un tipo anotación repetible sólo hay que anotar dicho tipo anotación con la meta-anotación predefinida `@Repeatable`. No obstante, antes de eso, es necesario que exista un “tipo anotación contenedor” para que el entorno de ejecución Java tenga alguna estructura donde guardar las múltiples anotaciones que se realicen sobre un mismo elemento.

Definimos un tipo anotación cualquiera que queremos convertir en repetible:

```
public @interface TipoAnotacionRepetible { ... }
```

Creamos su “tipo anotación contenedor” correspondiente, que servirá como estructura para albergar las múltiples instancias del tipo anotación original, normalmente siguiendo la convención de nombrarlo como el plural del tipo anotación original:

```
public @interface TiposAnotacionesRepetibles {
    TipoAnotacionRepetible[] value;
}
```

Finalmente, añadimos la meta-anotación `@Repeatable` a la definición del tipo anotación original que queríamos hacer repetible, y poniendo como argumento de `@Repeatable` la clase del “tipo anotación contenedor” que acabado de definir:

```
@Repeatable(TiposAnotacionesRepetibles.class)
public @interface TipoAnotacionRepetible { ... }
```

7.- ANOTACIONES PREDEFINIDAS.

Los siguientes tipos anotación predefinidos son de uso opcional en el sentido de que podemos crear aplicaciones Java sin hacer nunca uso de ellos. No obstante, su uso es recomendable dentro de un proceso de desarrollo que trate de seguir las mejores prácticas de programación y con casi total seguridad surgirán ocasiones en que nos convenga utilizarlos.

Tipos Anotación predefinidos sobre código (paquete java.lang)	
Anotación	Descripción
<code>@Deprecated</code>	Indica que el elemento de código anotado está “desfasado”, estando el uso de dicho elemento de código desaconsejado, ya que en el futuro será eliminado. Deberá proponerse en documentación una alternativa.
<code>@Override</code>	Indica que el método anotado redefine a otro ya existente.
<code>@SuppressWarnings (String[] value)</code>	Instruye al compilador a no arrojar warnings sobre el elemento de código anotado. Si el elemento incluye sub-elementos, los efectos de la anotación también se aplican a todos ellos. El valor o valores indican el tipo de warnings que se suprimen. La especificación del lenguaje cita dos valores: “ unused ” y “ deprecation ”, pero se pueden soportar muchos más por parte de los compiladores y los IDEs (como Eclipse).
<code>@SafeVarargs</code>	Instruye al compilador a no arrojar warnings en métodos de nº de argumentos variables (varargs) en caso de que la manipulación de los argumentos se haga de forma segura. Introducida en Java SE 7.
<code>@FunctionalInterface</code>	Tipo anotación de carácter informativo que indica que la declaración de tipo es una interfaz funcional (esto es: que tiene un único método abstracto, sin contar los de java.lang.Object). Introducida en Java SE 8.

7.1.- @Deprecated

@Deprecated	
Descripción	Anotación que permite marcar un elemento de código como desfasado, anticuado y a extinguir.
Clase	<code>java.lang.Deprecated</code>
Target	N/D, por defecto: aplicable sobre cualquier tipo de elemento de código fuente Java.
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 5, JSR 275 (A Metadata Facility for the Java™ Programming Language)
Signatura	<code>@Deprecated</code>
Elementos	N/D (anotación marcadora)

La anotación `@Deprecated` indica que el elemento de código anotado está “desfasado”, estando el uso de dicho elemento de código desaconsejado, ya que en el futuro será eliminado. Siempre que se use este tipo anotación sobre un elemento de código, debería indicarse en la documentación una alternativa válida que no esté desfasada. El uso común de esta anotación es indicar a los clientes de una API pública que deben dejar de utilizar algún elemento de dicha API (habitualmente métodos o clases, aunque se pueden anotar todos los tipos de elementos) debido a que un nuevo diseño o actualización de dicha API ha suprimido la necesidad del mismo.

Un ejemplo tradicional de clase “deprecada” (también se usa de forma muy común en español este neologismo) es el siguiente constructor de `java.util.Date` que era el que se usaba de forma más tradicional: `public Date(int year, int month, int date)`. Actualmente se encuentra “deprecado” en favor del constructor equivalente de la clase `java.util.GregorianCalendar` que le sustituirá en el futuro: `public GregorianCalendar(int year + 1900, int month, int date)`.

Cuando se usa un elemento “deprecado” el compilador arrojará un “warning”, un aviso de que estamos utilizando algo que no deberíamos utilizar, ya que está prevista su eliminación. Debería evitarse a toda costa utilizar elementos “deprecados”. No obstante, si aún así utilizamos código “deprecado” y queremos eliminar estos “warnings” podemos usar la anotación `@SupressWarnings` con el valor “deprecation”, como comentamos en su apartado más adelante.

Ejemplo: @Deprecated sobre un método

Para enfatizar el hecho de que los métodos deprecados no deberían utilizarse, algunos entornos de desarrollo, como Eclipse, marcan el nombre del método con un estilo de texto tachado:

```
@Deprecated
public static void metodoDesfasado(String mensaje) { ... }
```

7.2.- @Override

@Override	
Descripción	Anotación que permite señalar expresamente que un método concreto redefine a algún otro.
Clase	<code>java.lang.Override</code>
Target	<code>METHOD</code>
Retention	<code>SOURCE</code>
Otros	N/D
Desde	Java SE 5, JSR 275 (A Metadata Facility for the Java™ Programming Language)
Signatura	<code>@Override</code>
Elementos	N/D (anotación marcadora)

La anotación `@Override` indica que un método redefine la declaración de un tipo ascendente (o supertipo). Si un método está anotado con este tipo de anotación, el compilador arrojará un error a menos que se dé alguna de las siguientes condiciones:

- 1) El método redefine o implementa un método declarado en un supertipo.
- 2) El método tiene una signatura equivalente a algún método público de la clase `Object`.

Ejemplo 1: @Override sobre un método no redefinido → **error de compilación**

```
@Override
public String toString() { ... }
```

Esta declaración de redefinición del método `toString` de la clase `Object` arroja el siguiente error de compilación debido a que el nombre no está correctamente escrito (nótese la “s” minúscula): **The method `toString()` must override or implement a supertype method.**

Ejemplo 2: @Override sobre un método correctamente redefinido

```
@Override
public String toString() { ... }
```

7.3.- @SupressWarnings

@SupressWarnings	
Descripción	Anotación que instruye al compilador a suprimir warnings de diverso tipo sobre un elemento.
Clase	<code>java.lang.SuppressWarnings</code>
Target	<code>TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE</code>
Retention	<code>SOURCE</code>
Otros	N/D
Desde	Java SE 5, JSR 275 (A Metadata Facility for the Java™ Programming Language)
Signatura	<code>@SuppressWarnings ("unused")</code>
Elementos	<code>value = String[]</code> . Array de nombres que indican las categorías de warnings a suprimir.

La anotación `@SupressWarnings` indica al compilador que debe suprimir los tipos de warnings dados por su valor o valores (su tipo valor es un array).

`@SupressWarnings` se aplicará sobre el elemento anotado y, si este contiene otros sub-elementos de programa, sobre los sub-elementos contenidos en él. Este efecto es acumulativo. Es decir, por ejemplo: si se anota una clase para suprimir un tipo de warnings, como “unused” para las variables no utilizadas, y se anota un método de dicha clase para suprimir los warnings tipo “deprecation”, en el método se suprimirán ambos tipos.

Como convención, se recomienda utilizar esta anotación sobre el elemento más localizado donde sea efectivo, para que no se supriman más warnings de los estrictamente necesarios. Si, por ejemplo, si queremos suprimir los warnings en un método, deberíamos anotar ese método en lugar de anotar su clase entera.

El valor o valores del elemento indican el tipo de warnings que suprimirá el compilador. La especificación del lenguaje Java cita dos valores: “`unused`” y “`deprecation`” pero, de forma opcional, se pueden soportar muchos más valores por parte de los compiladores y los entornos de desarrollo, como ocurre en el caso de Eclipse, que ha añadido a su compilador interno muchos más valores para `@SupressWarnings`, como se puede ver en la siguiente página:
<http://www.thebuzzmedia.com/supported-values-for-suppresswarnings/>

Ejemplo 1: @SupressWarnings típico con “unused” sobre un método

En el siguiente ejemplo, todos los elementos que estén dentro del cuerpo del método (los sub-elementos del elemento de código anotado en este caso) y que no se usen, caso habitual de variables locales que se definen en algún momento mientras se escribe código, pero se acaban no utilizando y tampoco se borran, no generarán un warning acerca de que no están siendo utilizados en el código por parte del compilador.

```
@SuppressWarnings("unused")
public static void metodo() { ... }
```

Ejemplo 2: @SuppressWarnings típico con “unused” sobre una variable

En este ejemplo, el compilador no arrojará un error sobre el hecho de que no se esté utilizando la variable `variableNoUtilizada` pero, al no estar anotado el método, sí podría arrojar dicho warning sobre otras variables del resto del cuerpo del método, como es el caso de `otraVariableNoUtilizada` si efectivamente no está siendo utilizada para nada.

```
public static void metodo() {  
    @SuppressWarnings("unused")  
    int variableNoUtilizada = 0; // no arroja warning al estar anotada la variable  
  
    ...resto del cuerpo...  
  
    int otraVariableNoUtilizada = -1; // esta sí arroja warning al no estar anotada  
}
```

Ejemplo 3: @SuppressWarnings en varios elementos con efecto acumulativo

En el siguiente ejemplo se muestra cómo se puede acumular la supresión de varios tipos de warnings anotando sucesivamente elementos de código de niveles superiores y luego sucesivamente otros de niveles inferiores.

```
@SuppressWarnings("unused") // suprimir warnings de variables no usadas en toda la clase  
public class Clase {  
  
    @SuppressWarnings("deprecation") // suprimir warnings deprecated en todo el método  
    public metodo() {  
  
        // declaramos una variable que no vamos a usar y  
        // lo hacemos además con un constructor deprecado;  
        // esta línea no arroja ningún warning en el compilador  
        // gracias al efecto acumulativo de las sucesivas  
        // anotaciones @SuppressWarnings sobre la clase y el método  
        Date fechaNoUtilizada = new Date(113, 07, 31); // crea la fecha 31/08/2013  
    }  
}
```

Si no estuvieran las sucesivas anotaciones sobre la clase y el método, podríamos lograr el mismo efecto de supresión de warnings sobre la variable `fechaNoUtilizada` anotándola con la anotación multivaluada `@SuppressWarnings({ "unused", "deprecation" })`. No obstante, si no estuvieran las anotaciones sobre los elementos exteriores, otros elementos que no estuvieran directamente anotados sí que podrían arrojar warnings.

7.4.- @SafeVarargs

@SafeVarargs	
Descripción	Anotación que indica al compilador que en un método de argumentos variables (varargs) se maneja la lista de argumentos de forma segura desde el punto de vista de la seguridad de tipos.
Clase	<code>java.lang.SafeVarargs</code>
Target	<code>CONSTRUCTOR, METHOD</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 7
Signatura	<code>@SafeVarargs</code>
Elementos	N/D (anotación marcadora)

La anotación `@SafeVarargs` instruye al compilador a suprimir los warnings en métodos de número de argumentos variables (conocidos como *varargs*, de *variable arguments*) en caso de que, a juicio del programador, la manipulación de la argumentos de tipo genérico se haga de forma segura. `@SafeVarargs` sólo se puede colocar sobre métodos `static` o `final` y sirve para suprimir los warnings emitidos por el compilador sobre los métodos varargs con argumentos de tipo genérico cuando sospecha de [polución en el heap](#) debido al proceso de “borrado de tipos”.

`@SafeVarargs` impone que el compilador confíe en el programador (por lo cual, los diseñadores del lenguaje pensaron en llamar a este tipo anotación `@TrustMe`, “confía en mí”, lo cual hubiera permitido un uso más amplio del mismo, pero finalmente se decidió ceñirse a los métodos *varargs*). Este acto de fe puede ser incorrecto si el programador cree que el método es seguro con respecto a la polución del heap, pero realmente no es así. En ese caso, acabará saltando una excepción en tiempo de ejecución, probablemente una `ClassCastException` si el error efectivamente ha sido provocado por un manejo incorrecto de los datos de tipo genérico.

Ejemplo 1: @SafeVarargs colocada correctamente en un método seguro

Todos los elementos de la colección variable de argumentos se manejan en el cuerpo del método sin juguetear con sus tipos, realizando así una manipulación de forma realmente segura:

```
@SafeVarargs // indica que el método maneja los varargs genéricos de forma segura
public static <T> void metodoSeguroVarargs(Collection<? super T> colección, T...
elementos) {
    // procesamos los elementos de forma segura
    for (T elemento : elementos) { colección.add(elemento); }
}
```

Ejemplo 2: @SafeVarargs incorrectamente en un método NO seguro

Aquí el compilador no puede detectar una asignación ilegal y se provoca una `ClassCastException`. Este método, aunque está anotado como seguro con `@SafeVarargs`, realmente no lo es.

```
@SafeVarargs // NO realmente seguro; no se manejan los varargs genéricos de forma segura
protected static void metodoNoSeguroVarargs(List<String>... listaStrings) {
    Object[] arrayObjetos = listaStrings;
    List<Integer> listaTempEnteros = Arrays.asList(42);
    arrayObjetos[0] = listaTempEnteros; // incorrecto, pero compila sin warnings
    String string = listaStrings[0].get(0); // ClassCastException en tiempo ejecución
    System.out.println("metodoNoSeguroVarargs: " + string);
}
```

7.5.- @FunctionalInterface

@FunctionalInterface	
Descripción	Notación informativa que indica que la declaración de tipo es una interfaz funcional.
Clase	<code>java.lang.FunctionalInterface</code>
Target	<code>TYPE</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 8
Signatura	<code>@FunctionalInterface</code>
Elementos	N/D (anotación marcadora)

La anotación `@FunctionalInterface` es un tipo anotación de carácter informativo que sirve para indicar a los programadores que una interfaz determinada es una interfaz funcional.

Según la especificación del lenguaje Java para Java SE 8, ap. [9.8 Functional Interfaces](#): **una interfaz funcional es una interfaz que tiene un único método abstracto y, por tanto, representa un contrato de función única** (*single function contract*). Para determinar el método abstracto que constituye el contrato de función única no cuentan los métodos heredados de la clase `java.lang.Object` ni tampoco los métodos por defecto, ya que tienen una implementación y, por tanto, no pueden considerarse abstractos.

Al igual que pasaba con `@Override`, `@FunctionalInterface` puede ayudar a detectar errores de desarrollo, ya que si la clase anotada no es realmente una interfaz funcional, el compilador arrojará un error. No obstante, este tipo anotación es informativa, lo cual quiere decir que no es obligatoria su presencia para que una interfaz se considere interfaz funcional, ya que el compilador considerará como tal a cualquier interfaz funcional correctamente definida, independientemente de que la interfaz esté o no esté anotada con `@FunctionalInterface`.

El cometido fundamental de las interfaces funcionales es servir como tipos de referencia para la creación instancias en el entorno de las nuevas construcciones sintácticas introducidas en Java SE 8, como son las [expresiones lambda](#) y las [referencias a métodos y constructores](#).

Ejemplo: Anotación de una interfaz funcional con @FunctionalInterface

La interfaz `java.util.Comparator<T>` en Java SE 8 está anotada con `@FunctionalInterface` al ser una interfaz funcional. Tiene dos métodos abstractos, pero como uno es de la clase `Object` no cuenta y, por tanto, se ajusta a la definición de interfaz funcional.

```
@FunctionalInterface
interface Comparator<T> {
    boolean equals(Object obj); // este no cuenta porque es un método de Object
    int compare(T o1, T o2);
}
```

Al ser interfaz funcional, podemos crear implementaciones suyas mediante expresiones lambda:

```
// el 2º argumento de sort es una expresión lambda que ordena el array de Strings
Arrays.sort(arrayStrings, (String s1, String s2) -> s1.compareTo(s2));
```

8.- ANÁLISIS Y DISEÑO DE TIPOS ANOTACIÓN.

8.1.- Ventajas de las anotaciones.

Los principales beneficios por los que se añaden anotaciones (o cualquier otro tipo de mecanismo de inserción de meta datos) a un lenguaje es que **permiten ahorrar mucho código fuente redundante y repetitivo**.

Lo anterior además conlleva dos ventajas derivadas muy importantes:

- **(1) Facilitan una mayor legibilidad del código.**
- **(2) Hacen el código de las aplicaciones más fácil de mantener.**

En base a cómo suelen ser utilizadas habitualmente, **las anotaciones también facilitan el desarrollo de nuevas aplicaciones y su configuración**.

8.2.- Inconvenientes de las anotaciones.

Un análisis de las capacidades de las anotaciones exige que seamos verdaderamente críticos e imparciales, por lo que estaría incompleto si no se examinasen también los inconvenientes, carencias o posibles desventajas que podría conllevar el uso, mal uso y, especialmente, el abuso de las anotaciones.

Enunciaremos algunos de los principales inconvenientes que conlleva el uso de anotaciones de código en Java, y que principalmente tienen que ver con carencias o defectos en su diseño que desgraciadamente provocan que las anotaciones resulten a veces algo escasas en sus capacidades, rígidas o, al menos, incómodas de usar.

Entre los **inconvenientes más importantes de las anotaciones Java**, que podrían llevarnos a engaño o darnos algún problema, podemos encontrar los siguientes:

- Las anotaciones **no están orientadas a trabajar con datos**: ya que no son realmente metadatos (no son datos sobre datos), si no que son datos sobre otros elementos del lenguaje.
- Las anotaciones **no soportan herencia**: un tipo anotación no puede tener sub-tipos, lo cual es una desventaja importante a la hora de poder crear diseños más elegantes y compactos.
- Las anotaciones **no soportan valores de cualquier tipo**: sólo soportan valores de unos ciertos tipos muy concretos, lo cual dificulta en ocasiones su diseño, uso y funcionalidad.
- Las anotaciones **sólo soportan valores constantes en tiempo de compilación**: nada de llamadas a métodos, por lo que todo valor en una anotación siempre ha de ser constante.
- Las anotaciones **no soportan el valor null**: esto supone un grave inconveniente a la hora de diseñar procesadores de anotación que podrían tener interés en hacer uso de los valores de una lógica triestado, aprovechando el valor null; para ello se deben tratar de definir valores por defecto inválidos como -1 en caso de valores numéricos o la cadena vacía en el caso de cadenas.
- La anotaciones **no soportan el procesamiento sobre variables locales**. Un problema sorprendente y desconcertante tratándose de una tecnología tan importante como Java, así como del hecho de que tanto la Mirror API de J2SE 1.5 como la Pluggable Annotation Processing API de la JSR 269 están diseñadas para poder acceder a su información. Este inconveniente afortunadamente se ha resuelto, pero sólo a partir de Java SE 8.

8.3.- Cuándo usar anotaciones (y cuándo no).

¿Cómo saber si nos encontramos ante una situación adecuada para el uso de anotaciones? ¿Qué principios de diseño pueden emplearse para guiar nuestra decisión? Vamos a tratar de exponer los más importantes, de forma que podamos ser capaces de identificar los escenarios en los que el uso de las anotaciones resultará más apropiado.

El utilizar anotaciones cuando no debemos o diseñarlas de forma inadecuada puede provocar errores o inefficiencias que conlleven un esfuerzo extra a la hora de mantener el código. Lo ideal será conseguir diseñar tipos anotación que sean estables y que, si cambian, lo hagan al mismo tiempo que la semántica de los elementos que anotan.

Las anotaciones son un mecanismo sintáctico que puede aumentar la información de un determinado elemento del código fuente. Siendo así, están muy ligadas a su ubicación dentro del mismo, y es que dicha ubicación es una de sus propiedades y ventajas más importantes: **las anotaciones referencian directamente un elemento del código**.

Esto supone una enorme ventaja clave frente a otros componentes de la aplicación, como por ejemplo los ficheros de configuración, que han de utilizar los nombres cualificados (es.lcc.uma.PFCAnotaciones.NOMBREClase) para realizar dichas referencias. Este método tiene la desventaja de que es vulnerable a la refactorización y requiere por tanto un mayor esfuerzo de mantenimiento, mientras que con el uso de anotaciones todo resulta mucho más natural, rápido, cómodo y fácil de mantener.

Por ejemplo, puede anotarse una clase para declarar sus propiedades especiales de cómo debe ser persistida ([Hibernate](#)), cómo debe ser configurada vía un *framework* de [inyección de dependencias](#) ([Spring](#)), o podemos anotar un método para declarar su propósito como método de prueba ([JUnit](#)), de inicialización de las instancias de una clase ([@PostConstruct](#)) o de liberación de recursos de dichas instancias ([@PreDestroy](#)), etcétera.

No obstante, al diseñar tipos anotación debemos tener mucho cuidado de colocarlos cuidadosamente en su ubicación más adecuada, ya que, si no colocan adecuadamente, las anotaciones podrían hacer el código de una aplicación más difícil de mantener. Esto ocurre cuando las anotaciones son ubicadas en lugares incorrectos, o cuando introducen [acoplamiento](#).

Por tanto, debido a que, como hemos podido comprobar, las anotaciones también tienen sus inconvenientes y problemas, debemos recomendar prudencia y cautela a la hora de utilizarlas. Las anotaciones de código pueden resultar extremadamente útiles y eficientes en su tarea de facilitarnos las tareas de desarrollo de aplicaciones, mejorando la legibilidad del código, su calidad y facilitando su mantenimiento, pero hay que tener sumo cuidado, pararse a reflexionar y no usarlas a discreción a las primeras de cambio, puesto que si no las utilizamos adecuadamente para el fin para el que están ideadas, podrían llevarnos a un diseño y a un uso inefficiente e incluso incorrecto de las mismas.

8.4.- Análisis de tipos anotación.

Una vez decidido que se van a emplear un tipo anotación para alguna tarea en una aplicación, el proceso de análisis de un tipo anotación no se diferencia mucho del análisis de cualquier otro tipo de elemento software.

Para realizar el análisis de un tipo anotación se elabora un documento de requisitos, que puede estructurarse como un simple resumen de texto de los mismos, o de forma más elaborada y completa, como un catálogo de requisitos.

A partir de ahí, en base a los requisitos, nos centraremos en identificar los elementos que sería más apropiado definir, determinar sus tipos y, en caso de ser enumerados, los elementos a incluir en ellos.

Ejemplo: Detalle de requisitos para un tipo anotación @Nota (véanse los apartados de ejemplos de procesamiento para más detalles)

Requisitos:

Diseñar un tipo anotación que imprima por la salida del compilador, en el orden más similar posible al código fuente, notas de texto asociadas a cualquier tipo de elemento del lenguaje Java. El orden puede diferir del orden exacto del código fuente en caso de definirse clases anidadas.

Para ofrecer la posibilidad de extender la funcionalidad del tipo anotación, estas notas se podrán calificar en tipos: **NOTA** (por defecto), **DOCUMENTACION**, **TAREA**, **PENDIENTE**, **AVISO** y **ERROR**. Las notas de tipo **AVISO** y **ERROR** generarán un warning y un error respectivamente a la salida del compilador. Esto no tiene mucha utilidad práctica, si no que se incluye a efectos ilustrativos. Se deberán ofrecer opciones para incluir o excluir específicamente una lista de cualquiera de estos tipos de notas. Por defecto: incluir **TODO**, excluir **NADA**.

Se podrá establecer un valor de severidad de la nota respectiva relacionada con la importancia de su contenido: **NORMAL** (por defecto), **LEVE**, **GRAVE** y **FATAL**. Se deberán ofrecer opciones para incluir o excluir específicamente una lista de cualquiera de estas severidades. Por defecto: incluir **TODO**, excluir **NADA**.

Para permitir ordenar, si se desea, las diferentes notas atendiendo a diferentes criterios, el tipo anotación incluirá un campo de prioridad de tipo entero, donde la prioridad irá, en principio, de mayor a menor, por lo que la mayor prioridad vendrá dada por **Integer.MAX_VALUE**, la menor por **Integer.MIN_VALUE** y el valor por defecto será **0**. No obstante, deberá poderse configurar mediante las opciones pertinentes los rangos que queremos seleccionar (por defecto: el rango completo), así como si se ordenará de forma ascendente (por defecto) o descendente.

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Nota {

    // =====
    // ELEMENTOS
    // =====

    String texto() default "n/a";
    TipoNota tipo() default TipoNota.NOTA;
    SeveridadNota severidad() default SeveridadNota.NORMAL;
    int prioridad() default 0;

    // =====
    // TIPOS ENUMERADOS ESTÁTICOS
    // =====

    public static enum TipoNota { NOTA, DOCUMENTACION, TAREA, PENDIENTE, AVISO, ERROR };
    public static enum SeveridadNota { NORMAL, LEVE, GRAVE, FATAL };
    public static enum TipoOrden { CODIGO, TIPO, SEVERIDAD, PRIORIDAD };
    public static enum DireccionOrden { ASC, DESC };
}
```

8.5.- Diseño de tipos anotación.

Debido a lo simples que parecen en un principio, muchos desarrolladores no prestan demasiada atención a los criterios de diseño de las anotaciones cuando deciden usarlas. Muchas veces incluso se obvian las consideraciones más fundamentales del diseño software. Craso error, ya que, como cualquier otro elemento software, un mal diseño de los tipos anotación que usamos pueden desembocar en una aplicación que sea ineficiente o muy costosa de mantener.

En este apartado damos un breve repaso a algunas de las consideraciones y principios de diseño más importantes que hay que tener en cuenta cuando se está elaborando un diseño de tipos anotación.

1) Bajo acoplamiento.

Para mantener un bajo acoplamiento, **debemos tratar de mantener juntos los elementos que cambian juntos**, lo que trasladado a las anotaciones se traduce en que hay que tratar siempre de **poner las anotaciones sobre elementos que cambien al mismo tiempo que ellas**.

Además, debe procurarse siempre **respetar las relaciones de dependencia entre elementos**. Por ejemplo, si se quiere utilizar una anotación para expresar una dependencia, de deberá anotar únicamente el elemento dependiente (destino de la dependencia) no el dependido (origen de la dependencia), de forma que sean los objetos ubicados en el extremo final de la relación los que apunten al objeto origen, que no debe tener constancia en absoluto de dicha dependencia.

Imaginemos por ejemplo que utilizando un *framework* de inyección de dependencias queremos que se instancie la implementación por defecto de una interfaz y diseñamos un tipo anotación llamado `@ImplementacionPorDefecto` que indique cuál es dicha clase implementación por defecto de la interfaz:

```
@ImplementacionPorDefecto(ImplementacionInterfazEjemplo.class)
interface InterfazEjemplo { ... }
```

Aunque el tipo anotación `@ImplementacionPorDefecto` puede tener ciertos usos interesantes, como facilitar la ejecución de tests de unidad, su diseño viola el principio de que **hay que poner dependencias sólo en los elementos dependientes no en los dependidos** cuando se modela una relación de dependencia, puesto que, **si no se hace así, se introduce un acoplamiento totalmente indeseable entre los elementos**.

2) Anotar propiedades inherentes y tratar de evitar lo específico.

Si tenemos un servicio sin estado, podemos anotarlo por ejemplo con el tipo anotación `@Stateless` de Java EE, ya que el hecho de no tener estado es una propiedad inherente, una característica intrínseca, que emana exclusivamente de la naturaleza de dicha clase. Siendo así, dicha propiedad inherente no va a cambiar debido a un cambio en elementos externos.

```
@Stateless // recomendable, al ser una propiedad inherente al elemento anotado
interface EjemploServicioSinEstado { ... }
```

Sin embargo, hay otras propiedades que se pueden tratar de anotar mediante tipos anotación que no responden a propiedades inherentes de los elementos anotados, si no a casos de uso muy específicos.

Por ejemplo, siguiendo con la clase de servicio antes mencionada, a dicha clase le es indiferente desde donde puedan ser invocados sus métodos, ya sea de forma local o remota. Los servicios no son inherentemente o intrínsecamente locales o remotos. La decisión de acceder a ellos de forma local o remota no dependerá de los servicios, si no del caso de uso específico de cada uno de sus clientes externos. Especificar una anotación `@Local` o `@Remote` sobre la clase de servicio supondría estar anotando un elemento de código con unas características que no le son propias y que únicamente dependen de factores circunstanciales, además de que rompería la abstracción de la interfaz introduciendo información sobre la implementación del servicio.

```
@Local // no recomendable, al ser una propiedad impuesta por un caso de uso específico
// que no se corresponde con la naturaleza inherente del elemento anotado
@Stateless
interface EjemploServicioSinEstado { ... }
```

Como las propiedades inherentes emanan de la naturaleza del elemento son estables, al contrario de las propiedades específicas concretas que pueden variar en función de sus distintos casos de uso. Por tanto, si queremos anotaciones estables y fáciles de mantener, debemos, en la medida de lo posible, tratar de **anotar elementos con sus propiedades inherentes y evitar anotarlos con propiedades específicas de casos de uso concretos**.

3) Enfoque declarativo centrado en propiedades inherentes

En el principio anterior hemos recomendado tratar de usar las anotaciones para anotar las propiedades inherentes de los elementos anotados frente a las propiedades debidas a sus usos específicos y que deberían serles totalmente ajenos. Esto indica que el uso correcto de las anotaciones en muchos casos se orienta hacia un enfoque ciertamente declarativo.

Vamos a ilustrar este principio con un ejemplo muy común de mal diseño a la hora de construir un sistema de seguridad para las aplicaciones fácil de mantener. Supongamos que hemos desarrollado una aplicación bancaria en Java EE y queremos definir un sistema de seguridad. La propia plataforma Java EE nos proporciona el tipo anotación `@RolesAllowed`, perteneciente a la especificación `Common Annotations`, que nos permite enumerar el nombre de todos los roles de seguridad que pueden acceder a ejecutar código dentro del código anotado por dicha anotación. Por tanto, podemos anotar cualquier operación bancaria de la siguiente forma:

```
@RolesAllowed({"director", "auditor", "administrador"})
public void concederCredito(double cantidad) { ... }
```

El problema de este enfoque enumerativo en este caso es que provoca acoplamiento del servicio directamente con los roles definidos en algún otro módulo. Esto tiene la desgradable consecuencia de que cuando se quieran cambiar los roles de usuarios o la propia política de acceso habrá que revisar todas las anotaciones de seguridad para reflejar los cambios, lo cual introduce un esfuerzo enorme en el mantenimiento del código.

El uso de un enfoque enumerativo (más intuitivo y directo) en lugar de un enfoque declarativo es el origen de que este diseño acabe siendo muy problemático y endeble frente a cualquier mínimo cambio.

Podemos mejorar el diseño anterior volviendo a analizar la lógica de negocio para separar lo que es inherente de lo que no. Más que especificar la necesidad de algún tipo de política de seguridad simplemente enumerando los roles permitidos, podemos en su lugar declarar hechos (de ahí que lo llamemos estilo declarativo). En este caso, podemos elegir diseñar dos tipos anotación: uno que describirá el tipo de operación bancaria llevada a cabo por el elemento anotado (`@TipoOperacion`) y otro para el nivel de confidencialidad (`@NivelConfidencialidad`) en el acceso al mismo. Por tanto, el método anterior podría ahora quedar notado así:

```
@TipoOperacion(TiposOperaciones.CREDITOS)
@NivelConfidencialidad(NivelesConfidencialidad.ALTA)
public void concederCredito(double cantidad) { ... }
```

Nótese además que para no emplear directamente valores literales de caracteres o número, se han definido unos tipos enumerados para modelar de forma agrupada y bien diferenciada los tipos de operaciones (`TiposOperaciones`) y los niveles de confidencialidad (`NivelesConfidencialidad`) definidos por el nuevo diseño de nuestra aplicación de ejemplo.

Gracias a que ahora el elemento anotado lo está con tipos anotación que describen sus propiedades o características inherentes (que sabemos que son estables y no es muy probable que vayan a cambiar), el diseño es más fidedigno al modelo de negocio y el esfuerzo necesario para mantener el código frente a posibles cambios será pequeño o directamente nulo.

Además, ahora, las expresiones que asocian los elementos anotados con los roles de seguridad estarán vinculados también a propiedades inherentes de los elementos y, a ser posible, desligados de dichos elementos sobre los que aplican, como, por ejemplo, en ficheros de configuración independientes.

Con este diseño mejorado gracias al enfoque declarativo, gracias a que la información inherente expresada mediante las anotaciones ya define la mitad de la información sobre seguridad (ahora la información dada por las anotaciones tiene una fuerte carga semántica por sí sola), expresar las reglas que definen la política de seguridad de la aplicación se convierte en un proceso mucho más sencillo y natural.

Y si fuera necesario cambiar la política de acceso de un tipo de operación bancaria, algo que es externo a las propiedades inherentes de los elementos anotados, con este diseño mejorado sólo habría que tocar elementos verdaderamente externos: simplemente se trataría de actualizar las reglas de la política de seguridad dadas por los ficheros de configuración.

De esta forma, las anotaciones han otorgado a sus elementos anotados una verdadera identidad semántica, que nos ha facilitado el poder distinguirlos y trabajar con ellos desde el exterior, de una forma más desacoplada y, por tanto, más correcta desde el punto de vista de los principios generales del diseño software.

8.6.- Conclusiones.

Como hemos visto hasta ahora, las anotaciones son unos mecanismos sintácticos del lenguaje Java mediante los cuales puede dotarse a los elementos del código fuente del programa de información adicional que pueden traer muchos beneficios a nuestras aplicaciones. No obstante, las anotaciones también tienen sus defectos y, si no se utilizan correctamente, pueden conducir a diseños inefficientes o incorrectos.

Desde el punto de vista del diseño, hemos visto cómo las anotaciones son un mecanismo muy eficiente para declarar propiedades de los elementos del programa de forma directa sobre ellos mismos, favoreciendo lo que se conoce como [programación orientada a atributos](#).

Para poder ser capaces de elaborar el diseño que mejor saque partido de las capacidades de las anotaciones, debemos considerar el acoplamiento que pueden introducir si no se anotan exclusivamente las propiedades inherentes de los elementos anotados. Las anotaciones serán también mucho más estables (menos propensas a cambios) cuando informen de propiedades inherentes a los propios elementos anotados.

Si una clase no debe ser consciente de otra, sus respectivas anotaciones tampoco deberían ser conscientes entre ellas. De esta manera, todo estará correctamente desacoplado, sin dependencias innecesarias o incorrectas.

Siguiendo estos principios de diseño, las anotaciones se procurarán utilizar para declarar las propiedades inherentes de los elementos de código del programa. Para completar la lógica final de funcionamiento efectivo de una aplicación, la información dada por las anotaciones deberá ser complementada por los ficheros externos de configuración, *ad hoc* para cada caso de uso específico. De esta forma, cada mecanismo realiza la función más apropiada a su naturaleza y se mantiene el debido desacoplamiento entre elementos propios del programa y configuración.

9.- PROCESAMIENTO DE ANOTACIONES

9.1.- Conceptos básicos.

Una vez hayamos diseñado y definido un tipo anotación y anotado con él una cierta cantidad de elementos de código, ya estamos en condiciones de implementar un procesador que permita sacar provecho de dicho tipo anotación, ya que una anotación que no es procesada resulta poco más útil que un comentario.

Aunque nos centraremos en el procesamiento de anotaciones usando procesadores de anotaciones, el objetivo de este apartado es asentar los conceptos un poco más entre líneas que hay detrás del procesamiento de anotaciones y que no se suelen explicar claramente en la documentación oficial ni en los escasos artículos disponibles en Internet que abordan el tema del procesamiento de anotaciones.

Lo primero es tener claro que **existen diferentes métodos de procesamiento de anotaciones además de los típicos procesadores de anotaciones**. Todo mecanismo que permita examinar el código fuente o el *bytecode* de una clase Java compilada, leer la información relativa a las anotaciones presentes en ellos y procesarla, puede ser considerado un método de procesamiento de anotaciones.

Los principales **métodos de procesamiento de anotaciones** son los siguientes:

- 1) Compilador Java + Procesadores de anotaciones.**
- 2) Herramientas de manipulación de ficheros de clase.**
- 3) Código Java usando reflexión.**

Lo segundo que hay que tener claro es que, debido las diferencias entre cada una de las políticas de retención de las anotaciones, así como la propia naturaleza de los propios métodos de procesamiento, **no todos los métodos de procesamiento son aplicables a tipos anotación de todas las políticas de retención**.

9.2.- Procesamiento según la política de retención.

Como hemos visto en el apartado correspondiente a la meta-anotación **@Retention**, los tipos anotación pueden exhibir distintas políticas de retención: **SOURCE**, **CLASS** o **RUNTIME**. Cada una de ellas implica que la información acerca del tipo anotación debe “retenerse” (es decir: conservarse) hasta una determinada etapa.

Es muy importante comprender que los procesadores de anotaciones sólo están pensados y diseñados para procesar las anotaciones correspondientes a una política de retención **SOURCE**, es decir, a nivel del código fuente. Esto es así porque el encargado de iniciar el procesamiento de las anotaciones a través de los procesadores de anotaciones es el compilador.

Los procesadores de anotaciones nunca se lanzan explícitamente desde el código Java, sino que se diseñan para implementar una determinada interfaz y se configura el proceso de compilación para que el compilador sea quien los invoque.

Así pues, **según la política de retención, existen diversos métodos aplicables de procesamiento de anotaciones**. La relación es la siguiente:

Métodos de procesamiento de anotaciones según la política de retención del tipo anotación	
Política de retención	Métodos de procesamiento
SOURCE	Compilador Java + Procesadores de anotaciones.
CLASS	Compilador Java + Procesadores de anotaciones. Herramientas de manipulación de ficheros de clase.
RUNTIME	Compilador Java + Procesadores de anotaciones. Herramientas de manipulación de ficheros de clase. Código Java usando reflexión.

Es decir: con la política de retención **en tiempo de compilación (SOURCE)** la información sobre las anotaciones estará disponible directamente en el propio código fuente, y **dichas anotaciones podrán ser procesadas por el compilador Java y por los procesadores de anotaciones exclusivamente**.

Con la política de retención de **en el fichero de clase (CLASS)**, también **es posible que las anotaciones sean procesadas por el compilador configurando los procesadores de anotaciones**. Además, estando la información sobre las anotaciones contenida en los ficheros de clase, **se podrán procesar por otras vías si se dispone de la capacidad de leer la información acerca de las anotaciones del fichero de clase**. Existen diversas herramientas de manipulación de ficheros de clase que permiten realizar esto.

Finalmente, con la política de retención **en tiempo de ejecución (RUNTIME)**, también se podrá usar el compilador con los procesadores de anotaciones y las herramientas de manipulación de ficheros de clase. Asimismo, la información sobre las anotaciones será extraída de los ficheros de clase y cargada por la Máquina Virtual Java. Una vez cargada la definición de una clase, la API de reflexión nos dirá qué anotaciones están presentes en cada uno de los elementos de dicha clase. **Conociendo las anotaciones que anotan los elementos de las clases gracias a la API de reflexión, será posible procesar sus anotaciones directamente desde un código Java cualquiera, que no tendrá por qué ceñirse a los requisitos dados por la API de los procesadores de anotaciones**.

9.2.1.- Política de retención SOURCE.

Conceptualmente, la política de retención **en tiempo de compilación (SOURCE)** establece que la información sobre las anotaciones será retenida por el compilador únicamente en el propio código fuente. Tras finalizar la compilación, la información sobre las anotaciones con esta política de retención será descartada y no llegará a guardarse en los ficheros de clase generados.

Por ejemplo: las anotaciones @Override y @SupressWarnings, tienen una política de retención SOURCE, por lo que pueden afectar al compilador y hacer que este se comporte de forma diferente en base a su presencia, pero sólo podrán ser visibles, tenidas en cuenta y procesadas durante el proceso de compilación.

Las anotaciones con retención SOURCE son de ámbito exclusivo del código fuente y únicamente se podrán procesar durante la compilación del código, por lo que tenemos que tener claro que, al no guardarse la información sobre dichas anotaciones en los ficheros de clase compilados, **las anotaciones con política de retención SOURCE sólo serán procesables por los procesadores de anotaciones exclusivamente**.

9.2.2.- Política de retención CLASS.

Con la política de retención de **en el fichero de clase (CLASS)**, sabemos que **la información de las anotaciones se guarda en el bytecode de los ficheros de clase** compilados generados por el compilador Java (los ficheros con extensión .class).

A diferencia de la política de retención SOURCE, esta es ya **la primera política de retención cuyas anotaciones son visibles para el procesamiento de anotaciones**.

Esto es así porque **las APIs de procesamiento de anotaciones pueden leer la información sobre las anotaciones de esta política de retención precisamente desde el fichero de clase** previamente compilado durante el propio proceso de compilación, por lo que la información sobre dichas anotaciones está disponible para los procesadores a través de sus APIs de procesamiento.

Fuera de los procesadores de anotaciones, ya que la información de las anotaciones con política de retención CLASS no se carga por parte de la Máquina Virtual Java, sino que únicamente son guardados en los ficheros de clase, **sólo será posible procesar dichas anotaciones si se dispone de la posibilidad de leer la información sobre dichas anotaciones directamente desde los propios fichero de clase**.

Existen diversas herramientas de lectura y manipulación de ficheros de clase que permiten cargar la información contenida en los ficheros de clase generados por el compilador (ficheros con extensión .class que contienen el bytecode generado durante el proceso de compilación). Gracias a estas herramientas, se puede recuperar desde la definición de las clases contenidas en dichos ficheros, hasta las propiedades de sus miembros, como su paquete, constructores, campos, métodos, etcétera. **Tener la capacidad de leer la información sobre las clases y sus respectivas anotaciones** (de retención CLASS o RUNTIME, pero no SOURCE) directamente desde sus ficheros de clase correspondientes, **nos permitirá crear código que pueda procesar las anotaciones leídas según nuestras necesidades**.

9.2.3.- Política de retención RUNTIME.

Con la política de retención **en tiempo de ejecución (RUNTIME)**, al estar la información de las anotaciones guardada en los ficheros de clase, al igual que pasaba con la política de retención CLASS, será posible procesarlas mediante herramientas de manipulación de ficheros de clase. Además de ello, para los tipos anotación con política de retención RUNTIME **la información acerca de las anotaciones será cargada por el entorno de ejecución de la Máquina Virtual Java y estarán disponibles a través de la API de reflexión para la API de los procesadores de anotaciones**.

En cuanto al procesamiento de anotaciones, **la novedad que permite la política de retención RUNTIME es que permite procesar las anotaciones directamente haciendo uso de la API de reflexión del lenguaje Java, sin utilizar procesadores**.

IMPORTANTE: No obstante, hay que tener presente que mediante la API de reflexión sólo se pueden acceder y procesar las anotaciones con política de retención RUNTIME exclusivamente. La información de las anotaciones SOURCE no llegó al fichero de clase y la información de las anotaciones CLASS se queda en el fichero de clase y no es accesible a través de la API de reflexión. En el apartado dedicado al procesamiento de anotaciones en tiempo de ejecución mediante código Java usando reflexión trataremos de ilustrar este hecho con un ejemplo práctico.

9.3.- Métodos de procesamiento de anotaciones.

Como hemos visto en el apartado dedicado al procesamiento de anotaciones según la política de retención, existen diversos métodos de procesar las anotaciones disponibles sólo para las anotaciones de ciertos tipos de retención. Volvamos a examinar la tabla que ilustra dichos tipos de procesamiento y su disponibilidad según la política de retención de las anotaciones:

Métodos de procesamiento de anotaciones según la política de retención del tipo anotación	
Política de retención	Métodos de procesamiento
SOURCE	Compilador Java + Procesadores de anotaciones.
CLASS	Compilador Java + Procesadores de anotaciones. Herramientas de manipulación de ficheros de clase.
RUNTIME	Compilador Java + Procesadores de anotaciones. Herramientas de manipulación de ficheros de clase. Código Java usando reflexión.

Podemos ver como **el único método que permite procesar anotaciones de todas las políticas de retención es el método de los procesadores de anotaciones**, ya que estos tienen el privilegio exclusivo de recibir información del código fuente desde el compilador.

La utilización de herramientas de manipulación de los ficheros de clase generados es posible, lógicamente, en aquellas políticas de retención en las que se retenga la información acerca de las anotaciones en dichos ficheros de clase, esto es, para CLASS y RUNTIME.

El procesamiento de anotaciones mediante Código Java usando reflexión sólo está disponible para las anotaciones con política de retención RUNTIME exclusivamente.

9.3.1.- Compilador Java + Procesadores de anotaciones.

El **principal método de procesamiento de anotaciones** y el más habitual es el uso de los procesadores de anotación, que están precisamente diseñados para este propósito. Los procesadores de anotaciones son además **los únicos que pueden procesar anotaciones con retención SOURCE**, al tener la característica especial de poder comunicarse con el compilador y acceder a la “información privilegiada” que este puede proporcionarle de primera mano sobre los elementos del código fuente escrito que se pretende procesar.

Cuando el código Java es compilado, las anotaciones son procesadas por *plug-ins* del compilador llamados **procesadores de anotaciones**. Los procesadores pueden efectuar con el código fuente una gran variedad de acciones: desde lo más sencillo, como simplemente emitir mensajes informativos, hasta crear nuevos ficheros de recurso o incluso nuevos ficheros de código fuente, que procesados en sucesivas “rondas” del procesamiento.

El procedimiento habitual a seguir para procesar anotaciones mediante procesadores de anotaciones, debidamente ajustados a las especificaciones definidas por los diseñadores del lenguaje Java, depende de la versión de la plataforma Java SE sobre la que estemos desarrollando, ya que, como hemos comentado en la introducción histórica, cambió de forma sustancial de J2SE 1.5 a Java SE 6 con la definición de la especificación JSR 269.

Y es que es importante tener claro que los procesadores de anotación como tal son clases que implementan las interfaces oportunas y que serán procesadas por el compilador de Java (o por la herramienta apt en J2SE 1.5) con la adecuada configuración. Concretamente, para que una clase se convierta en procesador de anotaciones ha de implementar las siguientes interfaces (según la versión de la plataforma Java para la que se esté desarrollando):

Interfaces a implementar para que una clase funcione como procesador de anotaciones	
Versión de Java	Interfaz de los procesadores de anotaciones y métodos correspondientes
J2SE 1.5 (Sun Microsystems)	com.sun.mirror.apt.AnnotationProcessor Métodos: <code>void process()</code>
Java SE 6 (JSR 269)	javax.annotation.processing.Processor Métodos: <code>Set<String> getSupportedOptions() Set<String> getSupportedAnnotationTypes() SourceVersion getSupportedSourceVersion() void init(ProcessingEnvironment processingEnv) boolean process(Set<? extends TypeElement> annots, RoundEnvironment roundEnv) Iterable<? extends Completion> getCompletions(Element element, AnnotationMirror annotation, ExecutableElement member, String userText)</code>

Cada versión de la plataforma Java tiene sus propias particularidades a la hora de procesar las anotaciones, en función de si se trata de J2SE 1.5 y se utilizan procesadores de anotaciones junto con **apt**, o si, por otro lado, se está utilizando Java SE 6 y se trabaja con procesadores de anotaciones que siguen la especificación JSR 269. Aunque hay algunas similitudes entre ambas versiones, son muchas más las diferencias que las separan. En sucesivos apartados de este manual ahondaremos en el procesamiento de anotaciones según cada versión.

9.3.2.- Herramientas de manipulación de ficheros de clase.

Existen herramientas de manipulación de ficheros de clase (.class) que nos permiten extraer de ellos la información sobre la definición de la clase y las anotaciones que alberga su código; información que necesitaremos si queremos realizar nuestro propio procesamiento de anotaciones, sin necesidad de usar procesadores de anotaciones estándar o la API de reflexión.

Algunas de las APIs de manipulación de ficheros de clase más conocidas en la comunidad de desarrollo Java incluyen [Javassist](#), [BCEL](#), [ASM](#) o [cglib](#), entre [otras muchas existentes](#).

Como se comenta más adelante en el apartado “Información incompleta en el fichero de clase”, las anotaciones perteneciente a una clase se guardan el el fichero de clase en dos atributos: `RuntimeVisibleAnnotations` para las anotaciones de la clase que son accesibles en tiempo de ejecución vía reflexión (retención RUNTIME) y `RuntimeInvisibleAnnotations` para las que no son accesibles en tiempo de ejecución vía reflexión (retención CLASS). **Javassist**, por ejemplo, permite localizar un fichero de clase y cargarlo para recuperar esa información acerca de las anotaciones con un código como el siguiente:

Javassist – Código que permite recuperar las anotaciones RUNTIME y CLASS sobre un método

```
// classloader, nombre del resource y URI del fichero de clase a cargar
ClassLoader classLoacer = Thread.currentThread().getContextClassLoader();
String resourceClase = UsoAnotacionesUsuario.class.getCanonicalName().replace('.', '/') + ".class";
System.out.println("resourceClase = " + resourceClase);
URI ficheroClaseURI = classLoacer.getResource(resourceClase).toURI();
System.out.println("ficheroClaseURI = " + ficheroClaseURI);
FileInputStream ficheroInputStream = new FileInputStream(new File(ficheroClaseURI));

// cargamos el fichero de clase con Javassist y accedemos al método "inicializar"
ClassFile ficheroClassFile = new ClassFile(new DataInputStream(ficheroInputStream));
String nombreClase = ficheroClassFile.getName();
System.out.println("nombreClase = " + nombreClase);
MethodInfo metodoInicializar = ficheroClassFile.getMethod("inicializar");
System.out.println("metodoInicializar = " + metodoInicializar);

// anotaciones visibles en tiempo de ejecución vía reflexión (retención RUNTIME)
AnnotationsAttribute atribAnotacionesVisibles =
    (AnnotationsAttribute) metodoInicializar.getAttribute(AnnotationsAttribute.visibleTag);

if (atribAnotacionesVisibles != null) {
    for (javassist bytecode.annotation.Annotation a : atribAnotacionesVisibles.getAnnotations())
    {
        System.out.println("anotación RUNTIME > @" + a.getTypeName());
    }
}

// anotaciones invisibles en tiempo de ejecución vía reflexión (retención CLASS)
AnnotationsAttribute atribAnotacionesInvisible =
    (AnnotationsAttribute) metodoInicializar.getAttribute(AnnotationsAttribute.invisibleTag);

if (atribAnotacionesInvisible != null) {
    for (javassist bytecode.annotation.Annotation a : atribAnotacionesInvisible.getAnnotations())
    {
        System.out.println("anotación CLASS > @" + a.getTypeName());
    }
}
```

Si tenemos en cuenta que el método `inicializar` está anotado de la siguiente manera:

```
@AnotacionMarcadora  
@AnotacionElementoUnico(true)  
public static void inicializar() { ... }
```

El código anterior de recuperación de información de las anotaciones sobre dicho método devuelve la siguiente salida:

```
resourceClase = es/uma/lcc/PFCAnotaciones/ejemplos/anotaciones/uso/UsoAnotacionesUsuario.class  
ficheroClaseURI = file:/D:/PFC/Contenidos/3-Desarrollo/EclipseWorkspace/PFC-  
Anotaciones/bin/es/uma/lcc/PFCAnotaciones/ejemplos/anotaciones/uso/UsoAnotacionesUsuario.class  
nombreClase = anotaciones.ejemplos.uso.UsoAnotacionesUsuario  
metodoInicializar = inicializar ()V  
anotación RUNTIME > @anotaciones.ejemplos.definicion.AnotacionElementoUnico  
anotación CLASS > @anotaciones.ejemplos.definicion.AnotacionMarcadora
```

Como podemos deducir de la salida, en base a ser capaces de localizar la ruta del fichero de clase y cargarlo con Javassist, hemos podido extraer la información que necesitábamos del fichero de clase. En este caso, **no sólo hemos podido listar las anotaciones con retención RUNTIME, información que también nos hubiera dado la API de reflexión de Java, sino que además hemos podido listar las anotaciones con retención CLASS**, como es el caso de `@AnotacionMarcadora`. Esta última información sólo es posible acceder a ella con **herramientas de manipulación de ficheros de clase**.

Finalmente, hay que comentar que se han construido otras APIs a un nivel superior que pretenden facilitarnos la tarea de usar las APIs de bajo nivel anteriormente comentadas (y que son más complicadas de utilizar), para la localización de ficheros o definiciones de clase, así como la búsqueda y recuperación de información acerca de las anotaciones almacenada en los ficheros de clase.

Estas APIs de nivel superior son *wrappers* o adaptadores de las APIs de más bajo nivel, y que se centran en ofrecer una mayor comodidad y potencia de uso en las tareas de desarrollo más sencillas y propias del día a día. Un ejemplo bastante conocido de estas APIs más cómodas y potentes de usar es **reflections** [<http://code.google.com/p/reflections/>], que está construida usando **Javassist** como base.

9.3.3.- Código Java usando reflexión.

En este apartado vamos a ver los conceptos básicos acerca del acceso a la información sobre anotaciones a través de la API de reflexión. La API de reflexión del lenguaje Java reside en el paquete `java.lang.reflect` y, en cuanto a las anotaciones se refiere, con la aparición de las mismas en J2SE 1.5, se creó la interfaz `AnnotatedElement`, que permitía acceder en tiempo de ejecución de la Máquina Virtual Java a la información acerca de las anotaciones de tipos anotación con política de retención RUNTIME que hubieran sido aplicados sobre los distintos elementos de código de las clases cargadas en memoria por el entorno de ejecución.

java.lang.reflect.AnnotatedElement	
J2SE 1.5	<code>boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)</code> <code><T extends Annotation> T getAnnotation(Class<T> annotationClass)</code> <code>Annotation[] getAnnotations()</code> <code>Annotation[] getDeclaredAnnotations()</code>
Java SE 8	En Java SE 8 se incluyen, además de los anteriores, los siguientes métodos nuevos: <code><T extends Annotation> T[] getAnnotationsByType(Class<T> annotationClass)</code> <code><T extends Annotation> T getDeclaredAnnotation(Class<T> annotationClass)</code> <code><T extends Annotation> T[] getDeclaredAnnotationsByType(Class<T> annotationClass)</code>

Las implementaciones de la interfaz `AnnotatedElement` se usan en la práctica para representar elementos de código anotados en las clases cargadas por la Máquina Virtual Java. Así pues, como hemos dicho, a través de sus métodos, esta interfaz permite al programador “leer” o acceder reflexivamente a la información sobre las anotaciones de los tipos anotación que tengan una política de retención RUNTIME.

Concretamente, son las siguientes clases del paquete `java.lang.reflect`: `Package`, `Class`, `Constructor`, `Field` y `Method`, las que implementan la interfaz `AnnotatedElement`.

En Java SE 8 se añadió una **nueva clase implementadora**: `Parameter`, que modela los parámetros formales, lo cual permite realizar reflexión sobre los parámetros de métodos y constructores a través de la API de reflexión, algo no soportado anteriormente.

Además, en Java SE 8 también se añadieron varios métodos nuevos permitían que la interfaz `AnnotatedElement` añadir **soporte para anotaciones repetidas**.

Así pues, haciendo uso de la API de reflexión es posible recorrer las clases cargadas en tiempo de ejecución en la Máquina Virtual Java y consultar, a través de los métodos de la interfaz `AnnotatedElement`, qué elementos de código, dentro de la definición de dichas clases, están anotadas con anotaciones pertenecientes a tipos anotación exclusivamente con retención RUNTIME y, en base a ello, procesarlas como se deseé.

Este método de procesamiento de anotaciones, a pesar de ser el más limitado, también es el más inmediato y fácil de implementar, ya que está soportado directamente por las facilidades de una API estándar como es la API de reflexión de `java.lang.reflect` y no nos obliga a depender de librerías externas (como en el caso de las herramientas de manipulación de ficheros de clase) o implementar interfaz alguna (como los procesadores de anotaciones estándar). Es por esto que la mayoría de ejemplos de procesamiento de anotaciones que se ven por Internet no son usando procesadores de anotaciones como tal, si no usando la API de reflexión.

Lo que sigue es un sencillo ejemplo de cómo se puede usar la API de reflexión para comprobar si una clase y sus correspondientes métodos están anotados con una anotación determinada y, si es así, mostrar un mensaje informativo:

Ejemplo sencillo de uso de la API de reflexión para procesamiento de anotaciones

```
// cargamos la clase para acceder a sus anotaciones a través de la API de reflexión
Class clase = Class.forName(ClaseAnotadaHolaMundo.class.getCanonicalName());

// imprimimos su nombre
System.out.println("clase.getSimpleName() = " + clase.getSimpleName());

// consultamos si la clase como tal está anotada con la anotación que queremos procesar
boolean claseAnotada = clase.isAnnotationPresent(AnotacionHolaMundo.class);
System.out.println("clase.isAnnotationPresent(AnotacionHolaMundo.class) = " + claseAnotada);

// si la clase está anotada...
if (claseAnotada) {
    // recuperamos e imprimimos la anotación correspondiente a la clase
    // NOTA: el método getAnnotation devuelve un objeto que representará la anotación
    // de la clase AnotacionHolaMundo que se le pasa como parámetro si la clase "clase"
    // está realmente anotada con dicho tipo anotación o null en caso contrario;
    // en este caso sabemos que la anotación está presente, ya que lo hemos
    // consultado previamente a través del método isAnnotationPresent.
    Annotation anotacionClase = clase.getAnnotation(AnotacionHolaMundo.class);
    System.out.println("Anotación sobre clase -> "
        + anotacionClase.annotationType().getgetSimpleName());
}

// para cada uno de los métodos de la clase, listamos la anotación deseada, si la hay
for (Method metodo : clase.getDeclaredMethods()) {

    // recuperamos la anotación que queremos, si es que está anotando al método
    // NOTA: el método getAnnotation devuelve un objeto que representará la anotación
    // de la clase AnotacionHolaMundo que se le pasa como parámetro si el método "metodo"
    // está realmente anotado con dicho tipo anotación o null en caso contrario
    Annotation anotacionMetodo = metodo.getAnnotation(AnotacionHolaMundo.class);

    // si el método está anotando con la anotación que queremos...
    if (anotacionMetodo != null) {

        // imprimimos un mensaje informativo notificándolo
        System.out.println("Anotación sobre el método " + metodo.getName() + " -> "
            + anotacionMetodo.annotationType().getgetSimpleName());
    }
}
```

Nótese que primeramente se carga la definición de la clase en tiempo de ejecución con el método `class.forName`. Una vez hecho esto, ya podemos utilizar los métodos de la interfaz `AnnotatedElement` para recuperar toda la información que queramos sobre las anotaciones que anotan a esta clase u obtener una lista de sus elementos (como constructores, campos, métodos, etcétera) e interrogarlos igualmente sobre sus anotaciones correspondientes. En el ejemplo se ve como con la expresión `clase.isAnnotationPresent(AnotacionHolaMundo.class)` obtenemos un valor booleano que nos dice si la clase cargada está anotada o no con la anotación `AnotacionHolaMundo`. Si es así, con la expresión `clase.getAnnotation(AnotacionHolaMundo.class)` recuperamos un objeto `Annotation` que representa dicha anotación concreta e imprimimos sus propiedades. Luego hacemos lo mismo con los métodos que declara la clase. Se imprimirán aquellos que estén anotados con la anotación `AnotacionHolaMundo`.

El resultado de la ejecución del fragmento de código anterior, en el que suponemos que tanto la clase como el método interrogados están efectivamente anotados con el tipo anotación por el que se consulta, imprime la siguiente salida por consola:

Ejemplo sencillo de uso de la API de reflexión para procesamiento de anotaciones – Salida 1

```
clase.getSimpleName() = ClaseAnotadaHolaMundo
clase.isAnnotationPresent(AnotacionHolaMundo.class) = true
Anotación sobre clase -> AnotacionHolaMundo
Anotación sobre el metodo imprimirMensaje -> AnotacionHolaMundo
```

Para poner un ejemplo práctico que ilustre el hecho de que la API de reflexión no puede procesar las anotaciones que no tengan política de retención RUNTIME, imaginemos que modificamos la retención de `AnotacionHolaMundo` de `@Retention(RetentionPolicy.RUNTIME)` a `@Retention(RetentionPolicy.CLASS)`. La salida que obtendríamos en ese caso sería la siguiente:

Ejemplo sencillo de uso de la API de reflexión para procesamiento de anotaciones – Salida 2

```
clase.getSimpleName() = ClaseAnotadaHolaMundo
clase.isAnnotationPresent(AnotacionHolaMundo.class) = false
```

¿Qué ha ocurrido? Pues que al cambiar la política de retención del tipo anotación y ya no ser RUNTIME, la Máquina Virtual Java no dispone de información en tiempo de ejecución sobre las anotaciones de dicho tipo anotación. Por ello, aunque en el código la clase y el método estén efectivamente anotados con dicha anotación, la API de reflexión será incapaz de conocer ese hecho, puesto que esa información se ha perdido en el camino, ya que con la política de retención CLASS la información se ha guardado en el fichero de clase, pero ya no se ha retenido hasta el tiempo de ejecución y no es visible para la API de reflexión, dando un resultado que no es el que nos gustaría.

¿Cómo podemos solventar este problema? Hay una solución práctica muy sencilla y evidente para evitar este tipo de problemas: diseñar todos los tipos anotación con retención RUNTIME, si es que queremos procesarlas usando este método de la API de reflexión.

Diseñar todos sus tipos de anotación con una política de retención RUNTIME es lo que hacen muchos desarrolladores en la mayoría de los casos para quitarse de problemas. No obstante, si el tipo anotación a procesar viniera de una librería externa, y por tanto no pudiéramos influir en su diseño, si la política de retención de dicho tipo anotación no fuera RUNTIME y no fuera posible cambiarlo a RUNTIME, dicho tipo anotación no podría ser procesada nunca jamás usando el método de la API de reflexión, por lo que habrá que tener esta restricción muy en cuenta a la hora de tomar decisiones sobre el método de procesamiento de anotaciones a elegir.

10.- PROCESAMIENTO J2SE 1.5.

10.1.- Introducción.

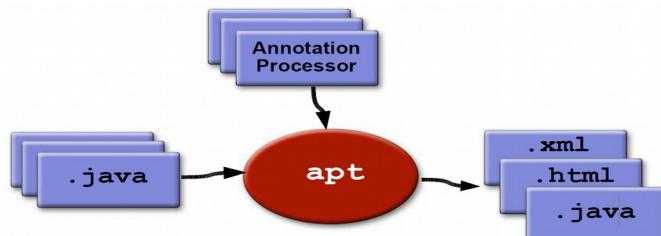
El **procesamiento de anotaciones en J2SE 1.5** fue introducido, poco después de que se aprobara la especificación **JSR 175** (A Metadata Facility for the Java™ Programming Language), por parte de Sun Microsystems. El equipo de ingenieros de Sun Microsystems, particularmente Joseph Darcy, introdujeron la noción de **procesadores de anotaciones**, la **Mirror API** (paquetes `com.sun.mirror`) y la **apt** (“annotation processing tool”, es decir: la “herramienta de procesamiento de anotaciones”).

IMPORTANTE: Actualmente, el **procesamiento de anotaciones en J2SE 1.5** usando **apt** está **deprecado y totalmente desaconsejado** en favor del **procesamiento de anotaciones usando JSR 269**. No obstante, lo incluimos como referencia histórica sin la cual este manual estaría claramente incompleto.

El **procesamiento de anotaciones en J2SE 1.5** consta de varios actores que trabajan juntos para implementar nuevas funcionalidades especiales en base a la meta-information añadida por las anotaciones a los elementos de código anotados. Este procesamiento, como ya sabemos, puede dar lugar a gran variedad de resultados interesantes.

Los **actores fundamentales del procesamiento de anotaciones en J2SE 1.5** son:

- (1) **ficheros .java**: ficheros de código Java anotado con ciertos tipos anotación.
- (2) **procesadores de anotaciones**: procesarán dichos tipos anotación sobre código anotado.
- (3) **apt**: herramienta de procesamiento que aplicará los procesadores al código anotado.



El procesamiento se basa en una **estrategia parecida a los preprocesadores** de otros lenguajes, donde **apt lleva a cabo el rol clásico de un preprocesador**, procesando el código inicial antes de ser enviado finalmente al compilador. Sin embargo, **a diferencia de un preprocesador tradicional, el preprocesamiento que realiza apt no es fijo, si no que viene dado por la lógica implementada por cada procesador de anotaciones**.

Para implementar la lógica de los procesadores de anotaciones, **es necesaria una API que modele los elementos del lenguaje Java a procesar (la Mirror API)**, así como tener en cuenta una serie de consideraciones especiales, como que, al existir la posibilidad de que pueda generarse nuevo código fuente durante el preprocesamiento, se hagan necesarias múltiples iteraciones de dicho preprocesamiento (las denominadas “rondas” de procesamiento de anotaciones). En los siguientes subapartados se explican en detalle todos los aspectos del procesamiento de anotaciones en J2SE 1.5.

10.2.- Cómo funciona el procesamiento J2SE 1.5.

Lanzamiento de apt mediante línea de comando

El procesamiento de anotaciones en J2SE 1.5 se realiza a través de la utilidad de línea de comando **apt** (annotation processing tool = herramienta de procesamiento de anotaciones). A través de las opciones de línea de comandos de **apt** se configura todo el procesamiento.

Típicamente se suelen indicar las mismas opciones que para **javac** (el compilador de Java) más algunas opciones adicionales específicas de **apt**, como el procesador de anotaciones a utilizar o, si son varios, las rutas donde buscar procesadores de anotaciones, el directorio de salida para los ficheros generados durante el procesamiento, los ficheros de código fuente que se desean procesar, etcétera.

Sin entrar en detalles (para lo cual puede consultarse el subapartado “Opciones de línea de comando de **apt**”), supongamos que iniciamos el procesamiento de anotaciones con una línea de comando de **apt** similar a la siguiente:

```
apt -cp CLASSPATH .\src\paquete\ClaseAnotada.java
```

En la línea de comandos se indica el classpath para la compilación, así como los **ficheros de código fuente sobre los que se desea realizar el procesamiento de anotaciones**.

Búsqueda de tipos anotación a procesar

Lo primero que hace **apt** es realizar una **búsqueda de tipos anotación** presentes en el código fuente que se va a procesar. En este caso, se analizará el texto del código fuente del fichero **ClaseAnotada.java** para encontrar qué anotaciones contiene. Supongamos que apt encuentra varias anotaciones del tipo anotación **@AnotacionEjemplo**.



Descubrimiento de factorías de procesadores de anotaciones

A continuación, **apt** lleva a cabo lo que se denomina el proceso de descubrimiento (en inglés: *discovery*) de las factorías de procesadores de anotación disponibles. Hay varias reglas que rigen este proceso; para más detalles véase el subapartado de funcionamiento de **apt**.



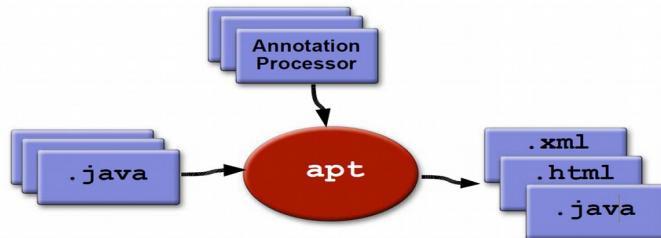
Emparejamiento factorías de proc. anotaciones ↔ tipos anotación

Una vez que **apt** ha determinado qué anotaciones tiene que procesar y con qué factorías de procesadores cuenta, realiza un emparejamiento entre factorías de procesadores y tipos anotación a procesar. Para ello, sigue el siguiente **procedimiento de emparejamiento**:

- (1) **apt** solicita a cada factoría una lista de los tipos anotación que procesan.
- (2) **apt** pedirá a la factoría un procesador si la factoría procesa alguno de los tipos anotación.
- (3) El proceso continúa hasta que todos los tipos anotación tienen una factoría asignada.

Ronda 1 de procesamiento de anotaciones

Una vez que se ha analizado los ficheros de código fuente, encontrado las anotaciones a procesar y emparejado sus tipos anotación con los procesadores de anotación que las procesarán, `apt` crea una instancia de cada procesador de anotaciones a ejecutar a través del método `getProcessorFor` de su factoría y a continuación llama al método `process()` del procesador para que este lleve a cabo el procesamiento correspondiente.



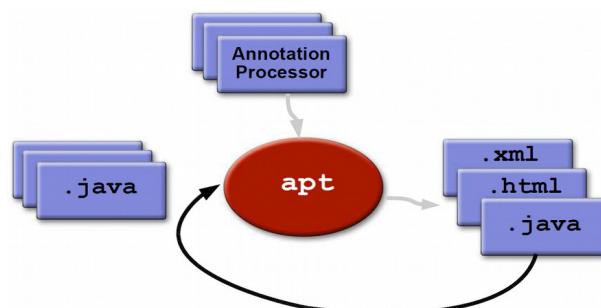
Cuando todos y cada uno de los procesadores de anotaciones descubiertos y emparejados han finalizado la ejecución de su método `process()`, termina la “ronda 1” o “primera ronda” del procesamiento de anotaciones. Además, en el procesamiento de anotaciones J2SE 1.5, al finalizar cada ronda se dispara el evento `roundComplete`, sobre las clases que se hayan registrado como `listener` del evento e implementen la interfaz `RoundCompleteListener`.

Rondas adicionales de procesamiento de anotaciones (si procede)

El concepto de “ronda” (del inglés: “round”) surge debido al hecho de que los procesadores de anotaciones pueden generar nuevo código fuente. Y dicho nuevo código fuente generado ¡¡ipodría a su vez contener anotaciones que es necesario procesar!!!

Es decir, que **mientras que haya procesadores de código que generen nuevo código fuente, el procesamiento de anotaciones no puede finalizar, debe continuar con una nueva “ronda de procesamiento” adicional**.

Si se hace necesaria una ronda de procesamiento adicional debido a que algún procesador de anotaciones ha generado nuevo código fuente, `apt` pasará a realizar un proceso muy similar al de la primera ronda, pero esta vez **lo que se tomará como argumento de entrada de código fuente a procesar será el nuevo código fuente recién generado en la ronda anterior**, como se ilustra en el siguiente esquema:

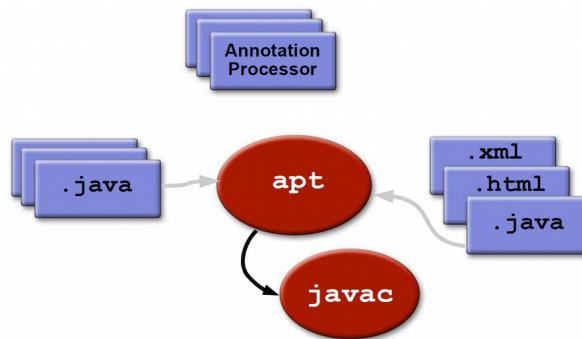


`apt` buscará las anotaciones existentes del nuevo código y emparejará los tipos anotación con las factorías de los procesadores adecuados de entre los disponibles, instanciándolos con el método `getProcessorFor` de su factoría y a continuación llamando al método `process()` de todos los procesadores para que procesen el nuevo código fuente generado en la ronda anterior.

Ronda final del procesamiento de anotaciones

Una vez se haya completado una ronda de procesamiento sin haber generado nuevo código fuente, se da por concluida la “ronda final” y se da por concluido todo lo que es el procesamiento de anotaciones como tal.

Terminado pues el procesamiento de anotaciones, **apt** realiza antes de terminar una llamada final a **javac** (el compilador de Java) para que compile todos los ficheros de código fuente, tanto los que había originalmente antes del procesamiento, como todos los nuevos ficheros de código que se hayan podido generar a lo largo de todas las rondas de procesamiento.



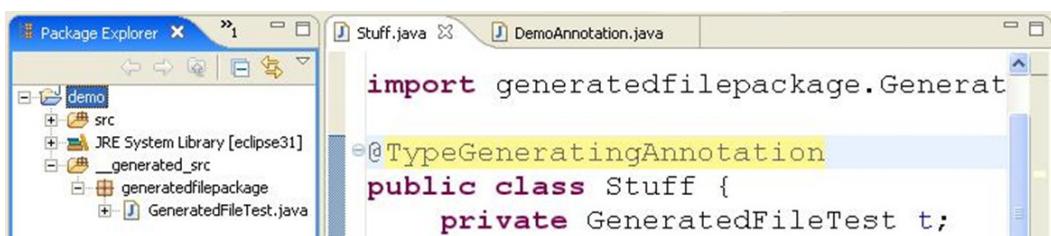
NOTA: La llamada final de **apt** al compilador Java puede no darse si se especifica la opción **-nocompile** en la línea de comandos. Esta opción, como veremos más adelante, puede tener mucho interés a la hora de depurar errores cometidos por los procesadores de anotaciones.

Resultados del procesamiento de anotaciones

Los resultados generados por el procesamiento de anotaciones pueden ser, como ya sabemos, de varios tipos, pero pueden agruparse en dos grandes categorías:

- (1) **Nuevos ficheros generados**, ya sea de código fuente u otro tipo de ficheros.
- (2) **Mensajes de compilación**, ya sean mensajes informativos, warnings o errores.

Los ficheros normalmente se generarán en el directorio que se haya configurado en **apt** como directorio de salida de nuevos ficheros usando su opción **-s DIRECTORIO**. Asimismo, si se usa procesamiento de anotaciones J2SE 1.5 en un IDE, como [Eclipse](#) o [NetBeans](#), aparecerán reflejados los nuevos ficheros en la vista correspondiente. La siguiente imagen muestra un nuevo fichero generado en el directorio configurado para ello dentro de un proyecto de [Eclipse](#):



Los mensajes arrojados por los procesadores de anotaciones durante el procesamiento podrán leerse directamente en la consola de comandos si se usa **apt** por línea de comandos, o en el caso de usar un IDE, igualmente en la vista que oportunamente establezca dicho IDE.

10.3.- Componentes del procesamiento J2SE 1.5.

10.3.1.- Procesadores de anotaciones J2SE 1.5.

10.3.1.1.- Procesadores de anotaciones.

Los procesadores de anotaciones J2SE 1.5 son clases que implementan la interfaz `com.sun.mirror.apt.AnnotationProcessor`, en cuyo método `process()` deberá residir la implementación del procesamiento de anotaciones. El procesador de anotaciones más básico que tiene sentido implementar, que sólo arroja un mensaje informativo al compilador sería:

```
public class AnotacionEjemploProcessor implements AnnotationProcessor {

    // =====
    // VARIABLES DE INSTANCIA
    // =====

    // entorno de procesamiento
    private final AnnotationProcessorEnvironment env;

    // =====
    // CONSTRUCTOR
    // =====

    public AnotacionEjemploProcessor(AnnotationProcessorEnvironment env) {

        // guardamos la referencia al entorno de procesamiento
        this.env = env;
    }

    // =====
    // MÉTODOS DE APLICACIÓN
    // =====

    public void process() {

        // imprimimos un mensaje informativo a la salida del compilador
        this.env.getMessager().printNotice("¡Hola, mundo!");
    }
}
```

Nótese como en el constructor se recoge y se guarda una referencia a la variable del entorno de procesamiento `AnnotationProcessorEnvironment`. Esto se hace así porque el entorno sólo se proporciona al procesador a través de su constructor y es una variable vital tener a mano, así que es común tenerla como variable de instancia del procesador.

CUIDADO: Este es un ejemplo para ilustrar lo mínimo que se le exige tener a un procesador de anotaciones, pero, obviamente, no es un ejemplo práctico. Normalmente, los procesadores, incluso para realizar tareas sencillas, guardarán como variables de instancia referencias, además de al entorno de procesamiento, a otras variables, como un `Messager` para emitir mensajes de compilación. Además, también será muy común que el constructor de los procesadores no reciba sólo el `AnnotationProcessorEnvironment`, si no también el conjunto de declaraciones de los tipos anotación a procesar `Set<AnnotationTypeDeclaration>`. Para más detalles, puede examinarse el código fuente de los ejemplos ilustrativos de este manual.

10.3.1.2.- Factorías de procesadores de anotaciones.

Las factorías de procesadores de anotaciones J2SE 1.5 son clases que implementan la interfaz **AnnotationProcessorFactory** del paquete **com.sun.mirror.apt** y cuya función principal es construir y suministrar nuevas instancias de procesadores de anotaciones.

Además, las factorías de procesadores de anotaciones J2SE 1.5 sirven de referencia para consultar las características de los procesadores de anotaciones disponibles durante la fase inicial de descubrimiento y emparejamiento de procesadores de anotaciones.

Métodos de la interfaz com.sun.mirror.apt.AnnotationProcessorFactory	
Método	Descripción
Collection<String> supportedAnnotationTypes ()	Devuelve una colección con el nombre canónico de las clases de tipos anotación soportados por las instancias de procesadores que construye la factoría.
Collection<String> supportedOptions ()	Devuelve una colección con las opciones de línea de comandos soportadas por los procesadores que construye la factoría. Cada opción empezará con -A, por ejemplo: “-Adebug”, “-AformatoSalida”, etc.
AnnotationProcessor getProcessorFor (Set<AnnotationTypeDeclaration> atds, AnnotationProcessorEnvironment env)	Devuelve una instancia de procesador de anotaciones construida por la factoría que procesa el conjunto de tipos anotación parámetro. El conjunto puede venir vacío en caso de un procesador que implemente todas las anotaciones (es decir: que declara que soporta “*”).

La factoría de procesadores de anotaciones J2SE 1.5 más básica que tiene sentido implementar, con soporte de un único tipo anotación y sin opciones de línea de comandos, sería:

```
public class ImprimirMensajeProcessorFactory implements AnnotationProcessorFactory {  
  
    public Collection<String> supportedAnnotationTypes() {  
  
        // nombres cualificados de los tipos anotación  
        // soportados por este procesador  
        String[] tiposAnotacionSoportados =  
            { ImprimirMensaje.class.getCanonicalName() };  
  
        // devolvemos los tipos soportados como una lista  
        return Arrays.asList(tiposAnotacionSoportados);  
    }  
  
    public Collection<String> supportedOptions() {  
  
        // devolvemos un conjunto vacío, ya que este  
        // procesador no soporta ninguna opción específica  
        return Collections.emptySet();  
    }  
  
    public AnnotationProcessor getProcessorFor(  
        Set<AnnotationTypeDeclaration> atds, AnnotationProcessorEnvironment env) {  
  
        // si el conjunto de tipos anotación viene vacío y  
        // este no es un procesador universal (que soporta "*"),  
        // es una buena práctica devolver el procesador identidad  
        if (atds.isEmpty()) return AnnotationProcessors.NO_OP;  
  
        // devolvemos una instancia del procesador correspondiente  
        return new ImprimirMensajeProcessor(atds, env);  
    }  
}
```

Uso de comodines * en la indicación de tipos anotación soportados

Como respuesta al método `supportedAnnotationTypes` se pueden devolver no sólo exactamente nombres canónicos de clases, si no también expresiones con comodines como “`nombre.paquete.*`” para indicar que el procesador soporta todos los tipos anotación no de ese paquete en particular, si no de todos los tipos anotación cuya clase empiece por dicho `nombre.paquete`. Esta es una forma de concentrar el procesamiento de los tipos anotación pertenecientes a un mismo paquete en un único procesador o, al menos, en una única factoría, ya que una factoría de procesadores de anotaciones podría elegir, según su implementación, construir instancias de diferentes procesadores según el tipo(s) anotación a procesar.

Procesadores de anotaciones universales

Las factorías también pueden especificar simplemente “*” para representar a “todos los tipos anotación”. Esto introduce el concepto de “**procesador universal**”: un procesador de anotaciones que es invocado siempre, aunque el código fuente a procesar no tenga anotaciones.

Los procesadores universales se conciben como procesadores especiales que pueden procesar el conjunto completo del código fuente para realizar una tarea general, en lugar de procesar sólo los elementos de código anotados. Los procesadores universales serán invocados incluso en ausencia total de anotaciones y a través de `getSpecifiedTypeDeclarations`, por ejemplo, podrían recuperar las declaraciones especificadas sobre las que se ha lanzado el procesamiento y realizar algún tipo de procesamiento general sobre dichas ellas y/o sus subdeclaraciones, independientemente de que estuvieran anotadas o no.

CUIDADO: Un procesador universal reclamará para sí mismo el procesamiento de todos los tipos anotación, por lo que, debido a como está implementado el proceso de descubrimiento, normalmente al descubrirse un procesador universal, `apt` dejará de buscar más procesadores y sólo ejecutará el primer procesador universal que encuentre. Por tanto, los procesadores universales deberían invocarse por separado con mucho cuidado utilizando la opción `-factory`.

Ejemplo: Procesador de anotaciones universal

Supongamos una factoría `UniversalProcessorFactory` que soporta “*”:

```
public Collection<String> supportedAnnotationTypes() {  
    // nombres cualificados de los tipos de anotación soportados por este procesador  
    String[] tiposAnotacionSoportados = { "*" }; // procesador universal  
  
    // devolvemos los tipos soportados como una lista  
    return Arrays.asList(tiposAnotacionSoportados);  
}
```

Imaginemos que en el `process()` del procesador `UniversalProcessor` correspondiente simplemente mostramos las declaraciones de “tipos especificados” del procesamiento, así:

```
public void process() {  
  
    Collection<TypeDeclaration> tiposEspecificadosDecl = env.getSpecifiedTypeDeclarations();  
    this.message.printNotice("tiposEspecificadosDecl "  
        + "[" + tiposEspecificadosDecl.size() + "] = " + tiposEspecificadosDecl);  
}
```

Finalmente, invocamos el procesamiento de anotaciones sobre una clase **ClaseVacia** que no tiene ninguna anotación, como por ejemplo: `public class ClaseVacia { }`

En un caso normal de este estilo, no se invocaría ningún procesador de anotaciones y, por tanto, no habría ningún mensaje a la salida del compilador. No obstante, como hemos definido nuestro **UniversalProcessor** como un procesador universal, a pesar de que **ClaseVacia** no tiene anotaciones, el procesador **UniversalProcessor** será invocado en la primera ronda, provocando que a la salida del compilador se muestre el siguiente mensaje informativo:

```
Note: tiposEspecificadosDecl [1] = [anotaciones.ejemplos.procesamiento.java5.pruebas.ClaseVacia]
```

Teniendo acceso a la declaración de **ClaseVacia**, el procesador universal podría realizar algún tipo de procesamiento general sobre ella, como análisis, búsqueda de errores, validación, estadísticas, etc. Este precisamente es el propósito de que existan los procesadores universales.

Ciclo de vida de las factorías y almacenamiento de información de estado

Las factorías de procesadores de anotaciones J2SE 1.5 tienen un ciclo de vida bastante particular que las convierte en las clases ideales para guardar la información de estado que puedan necesitar sus instancias.

Esto es así porque **apt** instancia las clases factoría al principio del procesamiento de anotaciones y conserva dichas instancias de factorías durante toda la duración efectiva del proceso de procesamiento anotaciones, lo cual no ocurre con los propios procesadores de anotaciones, que son re-instanciados nuevamente en cada ronda de procesamiento, lo cual impide que la información guardada en sus variables de instancia persista de una ronda a otra.

Por tanto, las factorías de procesadores son un lugar ideal para guardar información de estado en variables de clase (estáticas), a las que las distintas instancias de procesadores podrán acceder y manipular a lo largo de toda la duración del procesamiento de anotaciones.

Ejemplo: Factoría de procesadores que guarda información de estado en variables de clase

La factoría del procesador utilizado como ejemplo en el apartado dedicado a los Visitor que se dedicaba a computar la media estadística de métodos por clase guardaba la información de estado necesaria para su procesador en variables de clase:

```
public class AnotacionEjemploVisitorProcessorFactory implements AnnotationProcessorFactory {

    // =====
    // VARIABLES DE CLASE
    // =====

    // número de clases procesadas
    static int numeroClases = 0;

    // número de métodos procesados
    static int numeroMetodos = 0;

    // media de métodos por clase
    static double mediaMetodosPorClase = 0.0d;

    // ... resto del código de la factoría ...
}
```

10.3.1.3.- Descubrimiento de declaraciones a procesar.

Lo más habitual al implementar un procesador de anotaciones en el método `process()` es empezar recolectando las declaraciones con las que dicho procesador tiene que trabajar.

Cuando se invoca el procesamiento de anotaciones, se le da un conjunto de clases sobre las que tiene que operar. Esto es lo que se denomina “**tipos especificados**” (“*specified types*”). Luego está lo que se conoce como “**tipos incluidos**” (“*included types*”), que incluyen a los tipos especificados más los tipos anidados dentro de ellos.

Para recuperar las declaraciones a procesar se debe utilizar alguno de los dos métodos siguientes (pertenece ambos a `AnnotationProcessorEnvironment`):

1) `Collection<TypeDeclaration> getSpecifiedTypeDeclarations()`

`getSpecifiedTypeDeclarations` es el método más estándar, ya que devuelve la lista de declaraciones de los “tipos especificados” en la invocación del procesamiento de anotaciones. Este es el punto de partida más básico y fundamental para poder implementar un procesador de anotaciones, ya que es lo mínimo que necesitaremos para empezar a buscar las declaraciones anotadas con el tipo anotación a procesar. Y es que, a partir de esta lista de “tipos especificados”, el procesador podrá construir la lógica para ir recorriendo cada uno de ellos, examinando sus miembros (campos, métodos, constructores, clases anidadas, etcétera) y descubriendo cuáles miembros están anotados con el tipo anotación a procesar y procesarlos convenientemente.

```
public void process() {  
  
    // obtenemos las declaraciones de los tipos especificados  
    Collection<TypeDeclaration> tiposEspecifDecls = env.getSpecifiedTypeDeclarations();  
  
    // procesamos cada una de las declaraciones de los tipos especificados  
    for (TypeDeclaration typeDecl : tiposEspecifDecls) {  
  
        // buscamos declaraciones anotadas con el tipo anotación a procesar  
        // en la declaración de tipo y sus subdeclaraciones, si es que el  
        // tipo anotación es aplicable a otros tipos de elementos de código  
  
        // ¿declaración de tipo anotada?  
        if (typeDecl.getAnnotation(AnotacionPruebas.class) != null) {  
  
            // declaración de tipo anotada con el tipo anotación a procesar  
            // ... procesamiento de la declaración de tipo ...  
        }  
  
        // ¿subdeclaraciones de campos anotadas?  
        for (FieldDeclaration fieldDecl : typeDecl.getFields()) {  
  
            if (fieldDecl.getAnnotation(AnotacionPruebas.class) != null) {  
  
                // subdecl. de campo anotada con el tipo anotación a procesar  
                // ... procesamiento de la subdeclaración de campo ...  
            }  
        }  
  
        // ... resto de for para los demás tipos de subdeclaraciones ...  
    }  
}
```

En el código anterior, cada uno de los cuerpos de procesamiento, tanto de la declaración de clase como de sus distintos tipos de subdeclaraciones, se podrían ubicar en un Visitor, que podría ser un **DeclarationScanner** o un **SimpleDeclarationVisitor**, en función de si queremos o no que se visiten automáticamente las subdeclaraciones de las clases a procesar.

2) `Collection<Declaration> getDeclarationsAnnotatedWith(AnnotationTypeDeclaration a)`

`getDeclarationsAnnotatedWith` tiene la ventaja de que devuelve las declaraciones anotadas con un tipo anotación dado entre todas las declaraciones y subdeclaraciones de los “tipos incluidos”. Se devuelven todas las declaraciones de todo tipo de elementos de código que estén anotadas con el tipo anotación parámetro en un orden indeterminado. Este método es muy interesante, ya que, si lo utilizamos, nos ahorraremos el tener que “navegar” por la jerarquía de declaraciones buscando las declaraciones anotadas a procesar, como hemos visto que era necesario hacer cuando se partía de las declaraciones de “tipos especificados” dadas por el método `getSpecifiedTypeDeclarations`.

```
public void process() {  
  
    // declaración del tipo anotación a procesar  
    // para poder buscarla con getDeclarationsAnnotatedWith  
    AnnotationTypeDeclaration anotacionTypeDeclaration =  
        (AnnotationTypeDeclaration) env.getTypeDeclaration(  
            AnotacionPruebas.class.getCanonicalName());  
  
    // buscamos las declaraciones del tipo anotación a procesar  
    Collection<Declaration> declsAnotadas =  
        env.getDeclarationsAnnotatedWith(anotacionTypeDeclaration);  
  
    // procesamos cada una de las declaraciones anotadas  
    for (Declaration declAnotada : declsAnotadas) {  
  
        // procesamos cada una de las declaraciones anotadas según su tipo  
  
        if (declAnotada instanceof ClassDeclaration) {  
  
            // ... procesamiento de la declaración de clase ...  
  
        } else if (declAnotada instanceof FieldDeclaration) {  
  
            // ... procesamiento de la declaración de método ...  
  
        } else if ... // ... resto de if para los demás tipos de declaración ...  
    }  
}
```

En este código, cada uno de los cuerpos de procesamiento se podrían ubicar en un Visitor, que podría ser un **DeclarationScanner** o un **SimpleDeclarationVisitor**, en función de si queremos o no que se visiten automáticamente las subdeclaraciones de las clases a procesar.

`getDeclarationsAnnotatedWith` vendrá bien para el procesamiento de tipos anotación cuyas anotaciones puedan ser procesadas en el orden indeterminado con el que las proporciona este método. Esto no es posible siempre, ya que hay ocasiones en que la lógica de procesamiento de las subdeclaraciones de una clase está vinculada a su clase contenedora y, debido a ello, necesitaremos forzosamente ir procesando las anotaciones en bloques de clase por clase.

Si tenemos que ir procesando en bloques de clase por clase, lo más habitual será utilizar `getSpecifiedTypeDeclarations`, y partir de la lista de declaraciones de “tipos especificados”,

como ya hemos visto. No obstante, hay una vía intermedia si aún queremos ahorrarnos el “navegar” por las declaraciones: usar `getDeclarationsAnnotatedWith` para obtener la lista de todas las declaraciones de un tipo anotación, y filtrar con un filtro que nos deje sólo sus declaraciones de clase, quedándonos con las declaraciones de clase de entre los “tipos incluidos” anotadas con el tipo anotación a procesar. El código de esta vía intermedia utilizando un `DeclarationScanner` para visitar las subdeclaraciones de la clase, sería similar al siguiente:

```

public void process() {

    // declaración del tipo anotación a procesar
    // para poder buscarla con getDeclarationsAnnotatedWith
    AnnotationTypeDeclaration anotacionTypeDeclaration =
        (AnnotationTypeDeclaration)
    env.getTypeDeclaration(AnotacionPruebas.class.getCanonicalName());

    // buscamos las declaraciones del tipo anotación a procesar
    Collection<Declaration> declsAnotadas = env.getDeclarationsAnnotatedWith(anotacionTypeDeclaration);

    // filtramos las declaraciones anotadas para quedarnos
    // sólo con las declaraciones de clases anotadas...

    // creamos el filtro para las declaraciones de clase
    DeclarationFilter filtroDeclaracionesClase =
        DeclarationFilter.getFilter(ClassDeclaration.class);

    // filtramos y poblamos la lista de declaraciones de clases anotadas
    Collection<TypeDeclaration> declsClaseAnotadas = new ArrayList<TypeDeclaration>();
    filtroDeclaracionesClase.filter(declsAnotadas).addAll(declsClaseAnotadas);

    // procesamos cada una de las declaraciones de clase anotadas
    for (TypeDeclaration typeDecl : declsClaseAnotadas) {

        typeDecl.accept(DeclarationVisitors.getSourceOrderDeclarationScanner(
            new SimpleDeclarationVisitor() { // Visor de preprocesamiento

                @Override
                public void visitClassDeclaration(ClassDeclaration classDecl) {

                    // declaración de clase anotada con el tipo anotación a procesar
                    // ... procesamiento de la declaración de clase ...
                }

                // implementamos el procesamiento de las subdeclaraciones de la clase,
                // aunque, como no sabemos a ciencia cierta si están anotadas con
                // el tipo anotación a procesar, hay que comprobarlo antes

                @Override
                public void visitFieldDeclaration(FieldDeclaration fieldDecl) {

                    if (fieldDecl.getAnnotation(AnotacionPruebas.class) != null) {

                        // subdecl. de campo anotada con el tipo anotación a procesar
                        // ... procesamiento de la subdeclaración de campo ...
                    }
                }

                // ... resto de redefinición de métodos visitX
                // para los demás tipos de subdeclaraciones ...

            }, DeclarationVisitors.NO_OP)); // Visor de postprocesamiento
    } // for
} // process()
}

```

Como hemos explicado, el ejemplo de código anterior implementa la vía intermedia entre (1) traernos con `getSpecifiedTypeDeclarations` las declaraciones de “tipos especificados” y buscar nosotros las declaraciones anotadas “navegando” por la jerarquía de declaraciones y (2) traernos con `getDeclarationsAnnotatedWith` todas las declaraciones de anotaciones de cualquier tipo de elemento del tipo anotación a procesar en un orden indeterminado y así ahorrarnos tener que andar buscándolas.

Esta vía intermedia sería (3) traernos con `getDeclarationsAnnotatedWith` todas las declaraciones de anotaciones del tipo anotación a procesar, filtrándolas para quedarnos sólo con las declaraciones de las clases anotadas con el tipo anotación a procesar, para luego procesar en bloque clase por clase.

CUIDADO: Hay que tener en cuenta un detalle trascendental: tanto los procedimientos (1) como (2) tienen una cobertura total de las anotaciones a procesar, mientras que (3) no, a no ser que el tipo anotación a procesar sea una clase. Esto es así porque en (1) buscamos las anotaciones a procesar “navegando” a través del árbol de jerarquía de declaraciones completo, y en (2) el método `getDeclarationsAnnotatedWith` ya nos devuelve las declaraciones de todas las anotaciones a procesar. En (3) sólo vamos a analizar las clases anotadas con el tipo anotación a procesar, por lo que, si el tipo anotación es únicamente aplicable a clases, no habría problema, pero si fuera aplicable a otros tipos de elementos, podría darse un caso como el siguiente, en el cual las anotaciones que hay dentro de los elementos de código de la clase no serían procesadas al no estar anotada como tal la declaración de la propia clase (siendo entonces filtrada por el `filtroDeclaracionesClase` y, por tanto, no llegando a visitarse sus subdeclaraciones):

```
public class ClaseAnotadaPruebas1 {  
  
    // ======  
    // VARIABLES DE CLASE  
    // ======  
  
    @AnotacionPruebas  
    public String campo;  
  
    // ======  
    // MÉTODOS DE APLICACIÓN  
    // ======  
  
    @AnotacionPruebas  
    public void m1() {}  
}
```

Por este motivo, en base a la naturaleza del diseño del tipo anotación, así como de las características de la lógica a implementar en su correspondiente procesador de anotaciones y, sobre todo, de los elementos de código sobre los cuales el tipo anotación va a ser aplicado, el desarrollador de procesadores de anotaciones tendrá que elegir en cada caso cuál es la elección a seguir para lo que podríamos denominar “**estrategia de descubrimiento de anotaciones**”.

Así pues, a la hora de implementar la lógica del procesamiento de anotaciones en un procesador, primero debe elegirse aquella estrategia de descubrimiento de anotaciones que, teniendo siempre una **cobertura completa de las anotaciones que se desean procesar**, resulte más eficiente y con una estructura bien separada que sea cómoda de mantener, para lo cual los Visitor pueden resultar un complemento útil en ciertas ocasiones.

10.3.1.4.- Opciones para los procesadores desde línea de comandos.

En la documentación oficial de `apt` se describe que se pueden pasar opciones a los procesadores de anotaciones usando las opciones extendidas `-A` de la siguiente manera:

`-Aclave [=valor]`

Las opciones que comienzan por `-A` están, de hecho, reservadas para los procesadores de anotaciones. No son interpretadas por `apt`, si no que se ponen a disposición de los procesadores de anotaciones a través de la operación `Map<String, String> getOptions()` del entorno de procesamiento `AnnotationProcessorEnvironment`.

Por tanto, si invocamos el procesamiento de anotaciones con una línea de comandos que invoque `apt` incluyendo las siguientes opciones:

`-Aopcion1=valor1 -Aopcion2=valor2`

el siguiente código debería permitirnos recuperar los valores de dichas opciones:

```
// referencia a todas las opciones dadas por línea de comandos
Map<String, String> opcionesLineaComandos = env.getOptions();

// obtenemos los valores en String de las opciones soportadas por el procesador
String valor1 = opcionesLineaComandos.get("-Aopcion1"); // !!! devuelve null !!!
String valor2 = opcionesLineaComandos.get("-Aopcion2"); // !!! devuelve null !!!
```

CUIDADO: El comportamiento oficial descrito no se cumple en la práctica. Cuando se establecen opciones con `-Aopcion=valor`, en el `Map<String, String>` de opciones se debería guardar “`opcion`” como clave y “`valor`” como valor, pero no es así. Las expresiones no se parsean correctamente y se guarda “`opcion=valor`” como clave y siempre `null` como valor.

Este comportamiento erróneo está documentado como bug en varios reportes:

<https://bugs.openjdk.java.net/browse/JDK-6258929> getOptions doesn't return options as key/values
<https://bugs.openjdk.java.net/browse/JDK-6345271> apt tool does not parse -A options correctly
<https://bugs.openjdk.java.net/browse/JDK-6354725> Custom APT options not separated into name/value

No obstante, como se puede ver en la resolución de dichos reportes, no va a ser corregido, ya que se dice que la comunidad de desarrollo Java ya se ha adaptado al problema y precisamente el corregirlo podría traer problemas de compatibilidad (*sic*). Además, ya de paso, se aprovecha la ocasión para instar a los desarrolladores a migrar a la API de procesamiento de anotaciones JSR-269, donde este comportamiento sí está correctamente implementado.

Evidentemente, si hemos de trabajar con el procesamiento de anotaciones J2SE 1.5, habrá que lidiar con este bug. A continuación se describen un par de alternativas para poder pasar correctamente opciones a los procesadores de anotaciones desde línea de comandos.

Paso de opciones desde línea de comandos: alternativas correctas

Dado el comportamiento incorrecto de la implementación del paso de opciones a los procesadores de anotaciones de **apt**, existen un par de alternativas para realizar correctamente el paso de parámetros desde la línea de comandos a los procesadores de anotación: (1) utilizar propiedades estándar o (2) utilizar una función auxiliar para parsear nosotros el valor. Se recomienda la alternativa de la función auxiliar al mantener el uso del mecanismo estándar.

Paso de opciones desde línea de comandos: (1) Utilizar prop. estándar

Ya que las opciones de procesadores de anotaciones estándar **-Aclave[=valor]** fallan, podemos utilizar el mecanismo de las propiedades estándar del entorno de ejecución de la Máquina Virtual Java (Java Virtual Machine o, abreviando, JVM) para pasar dichas opciones a los procesadores de anotaciones.

Las propiedades estándar se pueden establecer en el comando de ejecución de una implementación de la Máquina Virtual Java (comando **java** o **javaw** en Windows) utilizando la notación **-Dpropiedad=valor**.

No obstante, **dicha notación tal cual no es válida para apt**, ya que **apt** no es un entorno de ejecución de la JVM. Para que dicha notación sea aceptada correctamente por **apt** tendrá que ir precedida por **-J**, para que así los parámetros que sigan a **-J** sean transferidos al lanzador Java de **apt** y estén disponibles para los procesadores de anotaciones.

Así pues, para pasar opciones desde línea de comandos como propiedades de la Máquina Virtual Java a los procesadores de anotaciones de **apt**, cada opción deberá seguir la notación:

-J-Dopcion[=valor]

Para pasar un par de opciones a los procesadores, tendremos pues que escribir una línea de comandos similar a la siguiente (los ^ son saltos de línea en los scripts **.bat** de Windows):

```
apt.exe -version -J-Dopcion1=valor1 -J-Dopcion2=valor2 ^
-cp ".\target\classes;.\lib\apt-mirror-api.jar" ^
-factory anotaciones.ejemplos.procesamiento.java5.pruebas.AnotacionPruebasProcessorFactory ^
.\src\main\java\anotaciones\ejemplos\procesamiento\java5\pruebas\PruebasMain.java
```

Estas opciones estarán disponibles como propiedades para los procesadores de anotaciones cuando estos se ejecuten. Los procesadores podrán recuperar su valor como un **String** a través del método **java.lang.System.getProperty(String propiedad)**. Se pasa como argumento el nombre de la propiedad y dicho método devuelve su valor. Así pues, para recuperar en el método **process()** de un procesador las opciones especificadas haríamos:

```
// obtenemos los valores en String de las opciones soportadas por el procesador
String opcion1 = System.getProperty("opcion1");
messager.printNotice("opciones del procesador: opcion1 = " + opcion1);
String opcion2 = System.getProperty("opcion2");
messager.printNotice("opciones del procesador: opcion2 = " + opcion2);
```

Dicho código imprimirá la salida correcta esperada:

```
Note: opciones del procesador: opcion1 = valor1
Note: opciones del procesador: opcion2 = valor2
```

Paso de opciones desde línea de comandos: (2) Utilizar una función auxiliar

Podemos utilizar una función auxiliar para recuperar el valor de las opciones de línea de comandos pasadas a los procesadores de anotación utilizando las opciones oficiales correctas **-Aclave[=valor]**. La función auxiliar implementada respeta el contrato oficial acerca del comportamiento esperado por si el bug fuera corregido en alguna implementación de la Máquina Virtual Java. En caso de que el procedimiento correcto falle, como será lo habitual, se implementa una lógica especial que localiza el valor de la clave incorrecta “**clave=valor**” y extrae el valor correcto de la opción como lo que hay después del carácter “=”:

```
public static String obtenerValorOpcion(String nombreOpcion, Map<String, String> mapOpciones) {  
  
    // valor resultado  
    String valorOpcion = null;  
  
    // primero, buscamos la opción con el nombre correcto  
    // (para respetar el comportamiento del contrato oficial previsto)  
    valorOpcion = mapOpciones.get(nombreOpcion);  
  
    if (valorOpcion == null) {  
  
        // si no hemos podido recuperar el valor de la opción,  
        // recorremos las claves de la lista buscando "nombreOpcion=" (con el =)  
        for (String claveOpcion : mapOpciones.keySet()) {  
  
            if (claveOpcion.startsWith(nombreOpcion + "=")) {  
  
                // el valor de la opción es lo que viene después del =  
                // en el nombre de la clave guardado en el mapOpciones  
                valorOpcion = claveOpcion.substring(claveOpcion.indexOf("=") + 1);  
                break;  
            }  
        }  
    }  
  
    // devolvemos el resultado  
    return valorOpcion;  
}
```

Utilizando esta función auxiliar estática, el valor de las opciones que queremos pasar a los procesadores pueden extraerse en su método **process()** con el siguiente código:

```
// referencia a todas las opciones dadas por línea de comandos  
Map<String, String> opcionesLineaComandos = env.getOptions();  
messager.printNotice("opcionesLineaComandos = " + opcionesLineaComandos);  
  
// obtenemos los valores en String de las opciones soportadas por el procesador  
String opcion1 = obtenerValorOpcion("-Aopcion1", opcionesLineaComandos);  
messager.printNotice("opciones del procesador: opcion1 = " + opcion1);  
String opcion2 = obtenerValorOpcion("-Aopcion2", opcionesLineaComandos);  
messager.printNotice("opciones del procesador: opcion2 = " + opcion2);
```

El código anterior imprime los valores esperados:

```
Note: opcionesLineaComandos = {-classpath=.\\target\\classes;.\\lib\\apt-mirror-api.jar, save-  
parameter-names=null, -Aopcion1=valor1=null, -Aopcion2=valor2=null, -version=null,  
-factory=anotaciones.ejemplos.procesamiento.java5.pruebas.AnotacionPruebasProcessorFactory}  
Note: opciones del procesador: opcion1 = valor1  
Note: opciones del procesador: opcion2 = valor2
```

Ejemplo: Paso de opciones desde línea de comandos con valores por defecto y validación

Vamos a realizar un ejemplo de paso de opciones desde línea de comandos con la lógica más habitual que se estilará en esos casos: comprobación de si las opciones han sido o no establecidas y, en caso negativo, darles un valor por defecto, así como la correcta validación de los valores establecidos en el caso de que una de las opciones tenga valor numérico.

Entre las alternativas que funcionan en la práctica de paso de opciones a los procesadores vamos a utilizar en este caso la de la función auxiliar por ser la que respeta el procedimiento del comportamiento oficial utilizando las opciones **-Aclave [=valor]** de la línea de comandos.

El siguiente código recupera los valores de las opciones soportadas por el procesador, los analiza para ver si es necesario establecer los valores por defecto o si hay que parsear un valor numérico, y finalmente muestra los valores finales de dichas opciones.

```
public void process() {  
  
    // OBTENCIÓN DE LOS VALORES DE LAS OPCIONES SOPORTADAS  
  
    // referencia a todas las opciones dadas por línea de comandos  
    Map<String, String> opcionesLineaComandos = env.getOptions();  
    messenger.printNotice("opcionesLineaComandos = " + opcionesLineaComandos);  
  
    // obtenemos los valores en String de las opciones soportadas por el procesador  
    String opcionString = obtenerValorOpcion("-AopcionString", opcionesLineaComandos);  
    messenger.printNotice("opciones del procesador: opcionString (leida) = " + opcionString);  
    String opcionIntStr = obtenerValorOpcion("-AopcionInteger", opcionesLineaComandos);  
    messenger.printNotice("opciones del procesador: opcionInteger (leida) = " + opcionIntStr);  
  
    // VALIDACIÓN DE LOS VALORES DE LAS OPCIONES SOPORTADAS  
  
    // opcionString: comprobamos si es null o cadena vacía,  
    // en cuyo caso se establecerá el valor por defecto  
    if ((opcionString == null) || (opcionString.length()==0)) {  
        opcionString = ";Hola, mundo!"; // valor por defecto  
    }  
  
    // opcionInteger: comprobamos si es null o cadena vacía,  
    // en cuyo caso se establecerá el valor por defecto;  
    // si no, parseamos el valor dado a número y arrojamos  
    // una excepción si no fuera un valor entero válido  
    int opcionInteger = 1; // valor por defecto  
    if ((opcionIntStr != null) && (opcionIntStr.length()>0)) {  
        try {  
            opcionInteger = Integer.parseInt(opcionIntStr);  
        } catch (NumberFormatException nfe) {  
            messenger.printError(  
                "opciones del procesador: opcionInteger: valor incorrecto: " + nfe);  
        }  
    }  
  
    // INFORMACIÓN DE LOS VALORES FINALES DE LAS OPCIONES SOPORTADAS  
  
    // mostramos como mensaje informativo (que puede ser muy útil para depuración)  
    // los valores finales de las opciones soportadas por el procesador  
    messenger.printNotice("opciones del procesador: opcionString (final) = " + opcionString);  
    messenger.printNotice("opciones del procesador: opcionInteger (final) = " + opcionInteger);  
  
    // ... resto del procesamiento ...  
}
```

Ahora, si se ejecuta la siguiente línea de comandos, sin ninguna opción establecida:

```
apt.exe -version ^
-cp ".\target\classes;.\lib\apt-mirror-api.jar" ^
-factory anotaciones.ejemplos.procesamiento.java5.pruebas.AnotacionPruebasProcessorFactory ^
.\src\main\java\anotaciones\ejemplos\procesamiento\java5\pruebas\PruebasMain.java
```

el procesador nos muestra la salida correcta esperada sin haber podido leer las opciones desde la línea de comandos y, por tanto, asignando sus respectivos valores por defecto:

```
Note: opcionesLineaComandos = {save-parameter-names=null,
-classpath=.\\target\\classes;..\\lib\\apt-mirror-api.jar, -version=null,
-factory=anotaciones.ejemplos.procesamiento.java5.pruebas.AnotacionPruebasProcessorFactory}
Note: opciones del procesador: opcionString (leida) = null
Note: opciones del procesador: opcionInteger (leida) = null
Note: opciones del procesador: opcionString (final) = ;Hola, mundo!
Note: opciones del procesador: opcionInteger (final) = 1
```

Y, lo más importante, si se ejecuta la línea de comandos con las opciones establecidas:

```
apt.exe -version -AopcionString="Valor String" -AopcionInteger=123 ^
-cp ".\target\classes;.\lib\apt-mirror-api.jar" ^
-factory anotaciones.ejemplos.procesamiento.java5.pruebas.AnotacionPruebasProcessorFactory ^
.\src\main\java\anotaciones\ejemplos\procesamiento\java5\pruebas\PruebasMain.java
```

el procesador nos muestra la salida correcta esperada, asignando a cada opción los valores establecidos en línea de comandos:

```
Note: opcionesLineaComandos = {-classpath=.\\target\\classes;..\\lib\\apt-mirror-api.jar,
save-parameter-names=null, -AopcionString=Valor String=null, -version=null,
-factory=anotaciones.ejemplos.procesamiento.java5.pruebas.AnotacionPruebasProcessorFactory,
-AopcionInteger=123=null}
Note: opciones del procesador: opcionString (leida) = Valor String
Note: opciones del procesador: opcionInteger (leida) = 123
Note: opciones del procesador: opcionString (final) = Valor String
Note: opciones del procesador: opcionInteger (final) = 123
```

Finalmente, si como valor de opcionInteger no se establece un valor numérico correcto, por ejemplo, con **-AopcionString="Valor String" -AopcionInteger="Otro String"**, obtenemos también el comportamiento esperado: se arroja la excepción NumberFormatException y se envía al compilador el mensaje de error esperado, provocando un error de compilación que detiene todo el proceso para que así el valor de la opción numérica pueda corregirse:

```
Note: opcionesLineaComandos = {-AopcionInteger=Otro String=null,
-classpath=.\\target\\classes;..\\lib\\apt-mirror-api.jar, save-parameter-names=null,
-AopcionString=Valor String=null, -version=null,
-factory=anotaciones.ejemplos.procesamiento.java5.pruebas.AnotacionPruebasProcessorFactory}
Note: opciones del procesador: opcionString (leida) = Valor String
Note: opciones del procesador: opcionInteger (leida) = Otro String
error: opciones del procesador: opcionInteger: valor incorrecto:
java.lang.NumberFormatException: For input string: "Otro String"
Note: opciones del procesador: opcionString (final) = Valor String
Note: opciones del procesador: opcionInteger (final) = 1
1 error
```

Nótese como, con la lógica construida en el ejemplo, puesto que el valor dado por línea de comandos para **opcionInteger** era incorrecto, se ha establecido a su valor por defecto.

10.3.1.5.- Distribución de procesadores J2SE 1.5 en ficheros .jar.

Como veremos más adelante, el procesamiento de anotaciones consta de varias fases, muy similares a las fases de todo desarrollo de software: análisis, diseño, implementación, anotación de código cliente y obtención de resultados.

Cuando se termina de desarrollar un procesador de anotaciones, puede que queramos distribuirlo a terceras partes para que puedan utilizarlo. En este subapartado vamos a describir las formas más habituales y las mejores prácticas a la hora de preparar la distribución de los procesadores de anotaciones que desarrollemos.

Lo primero a señalar es que **un procesador de anotaciones Java se distribuye de forma muy similar a cualquier otra aplicación Java: en uno o varios ficheros .jar (Java ARchive)**. Que sean uno o más ficheros .jar dependerá del mayor o menor nivel de fragmentación por el que opte el diseñador del procesador de anotaciones.

Los archivos .jar son ficheros comprimidos en formato ZIP, por lo que cualquier editor de ficheros ZIP, como el libre [7-Zip](#), permite visualizar sus contenidos. Dentro de los archivos .jar suelen encontrarse ficheros compilados de clase y, a veces, también los ficheros de código fuente correspondientes. Las aplicaciones Java ejecutables normalmente definen una clase como punto de entrada para ejecutar su método main en el “fichero de manifiesto”, situado en la ruta **META-INF/MANIFEST.MF**. En el caso de los procesadores esto no tiene ningún sentido, ya que **los procesadores de anotaciones no son aplicaciones Java al uso**.

Ficheros habituales en la distribución de procesadores de anotaciones		
Fichero	Ejemplo	Descripción
Tipo(s) anotación	<code>TipoAnotacion1.class</code> , <code>TipoAnotacion2.class</code>	Clases de los tipos anotación definidos. Es necesario que estén en las rutas de búsqueda de clases para que puedan ser reconocidos por el compilador y/o el entorno de ejecución.
Factoría(s)	<code>ProcesadorAnotacion1Factory.class</code> <code>ProcesadorAnotacion2Factory.class</code>	Clases factoría de los procesadores definidos. Es necesario que estén en las rutas de búsqueda de factorías de procesadores para que <code>apt</code> los pueda descubrir y utilizarlas.
Fichero de descubrimiento de servicios (SPI)	Este es siempre el mismo y siempre está ubicado en la misma ruta canónica: <code>META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory</code>	Este fichero es el que utiliza <code>apt</code> para descubrir factorías de procesadores de anotaciones si no se utiliza su opción <code>-factory</code> para forzar la utilización de una única factoría concreta.

Que sean uno o más ficheros .jar los utilizados para la distribución de los ficheros de nuestros procesadores de anotaciones se debe a que se puede optar por cualquiera de estas vías:

- (1) Colocar todos los ficheros en un único el fichero .jar por comodidad.
- (2) Separar los tipos anotación y las factorías de procesadores en dos o más ficheros.

Esta última opción es considerada más apropiada por muchos desarrolladores, ya que separa dos conceptos claramente distintos. En algunos casos, en función de cómo se quisieran estructurar las dependencias de código podrían ser necesarias sólo las definiciones de los tipos anotación, pero no las factorías si no se va a realizar procesamiento de anotaciones. Esto permite mayor libertad a la hora de configurar las dependencias de una aplicación, además de que reduce el acoplamiento de los tipos anotación con sus factorías, lo cual, como decimos, puede ser deseable en algunos casos y hace que algunos desarrolladores opten por esta opción.

El fichero `META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory` sólo es necesario que se incluya en el fichero `.jar` donde se alojen las factorías de procesadores de anotaciones para que puedan ser descubiertas por `apt`. Puede no ser necesario si siempre se invoca `apt` con su opción `-factory`, que anula el proceso de descubrimiento de factorías de procesadores. Para más información sobre el descubrimiento de factorías de procesadores, puede consultarse el subapartado “Funcionamiento de `apt`”.

Supongamos que se opta por distribuir los ficheros de procesamiento de anotaciones del ejemplo de la tabla anterior en 2 ficheros `.jar`: uno para las definiciones de los tipos anotación y otro para las definiciones de las factorías de procesadores, con su respectivo fichero de descubrimiento ([SPI](#) = Service Provider Interface). La estructura de los ficheros `.jar` debería ser:

anotaciones-tipos.jar

```
nombre/canonico/paquete/anotacion1/TipoAnotacion1.class  
nombre/canonico/paquete/anotacion2/TipoAnotacion2.class
```

anotaciones-factorias.jar

```
nombre/canonico/paquete/factoria1/ProcesadorAnotacion1Factory.class  
nombre/canonico/paquete/factoria2/ProcesadorAnotacion1Factory.class  
META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory
```

El fichero `META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory` deberá contener una línea de texto por cada factoría de procesadores de anotaciones. Cada línea será el nombre canónico de cada una de las clases factorías contenidas en el fichero `.jar`. Para el caso del fichero `anotaciones-factorias.jar` tenemos 2 factorías, por lo que su contenido será:

Fichero <code>META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory</code>
<code>nombre.canonico.paquete.factoria1.ProcesadorAnotacion1Factory</code>
<code>nombre.canonico.paquete.factoria2.ProcesadorAnotacion2Factory</code>

Si optamos por distribuir nuestros ficheros de procesamiento de anotaciones en un único fichero `.jar`, lo único que habrá que hacer es combinar la estructura descrita para ambos ficheros en uno solo.

NOTA: Los ficheros de procesamiento de anotaciones exportados en los ficheros `.jar` no tienen por qué estar necesariamente en dichos ficheros `.jar`. Pueden estar simplemente en el `CLASSPATH` o el `FACTORYPATH`. Lo importante es que `apt` pueda descubrirlos. No obstante, a la hora de distribuirlos a terceros, el procedimiento habitual es empaquetarlos en ficheros `.jar`.

Una vez que tengamos los ficheros de procesamiento de anotaciones bien empaquetados en nuestros ficheros `.jar` ya estaremos en disposición de distribuirlos a quienes queramos. No obstante, ya hemos dicho que los procesadores de anotaciones no son aplicaciones normales y no se ejecutan como tales (con el clásico `java -jar aplicacion.jar`), si no que se ejecutan a través de `apt`. Para saber cómo exactamente ha de realizarse la invocación a `apt` para iniciar el procesamiento de anotaciones, pueden consultarse los subapartados “Funcionamiento de `apt`” y “Opciones de linea de comandos de `apt`”.

10.3.2.- Mirror API.

El procesamiento de anotaciones J2SE 1.5 necesitaba una API sobre la que basarse y que pudieran utilizar los procesadores de anotaciones, para lo cual se creó la [Mirror API](#), que se podría describir en pocas palabras como **una API equivalente a la API de reflexión, pero en tiempo de compilación**. De hecho, su nombre es un juego de palabras por la dualidad API de reflexión (en tiempo de ejecución) → reflejo ↔ espejo → Mirror API (tiempo de compilación).

Las **funciones básicas de la Mirror API** y de los elementos que la componen son tres, a saber: (1) establecer la arquitectura general de funcionamiento de los procesadores de anotaciones, (2) definir las interfaces de comunicación entre los procesadores y la herramienta externa de procesamiento, y (3) proporcionar un modelo de representación lo más completo posible de los elementos de código del lenguaje de programación Java, que será utilizado por los procesadores de anotaciones para describir su lógica interna.

Paquetes de la Mirror API (com.sun.mirror)	
Paquete	Descripción
com.sun.mirror.apt	Arquitectura general de procesamiento. Comunicación procesadores ↔ apt.
com.sun.mirror.declaration	Modelado del lenguaje Java. Declaraciones de elementos de código fuente.
com.sun.mirror.type	Modelado del lenguaje Java. Tipos simbólicos.
com.sun.mirror.util	Clases de utilidad para el procesamiento. Clases e interfaces Visitor.

La Mirror API proporciona una visión de la estructura de un programa Java en tiempo de compilación, basado en el código fuente escrito y con un acceso de sólo lectura. El proceso es el siguiente: (1) se lee el fichero de código fuente .java escrito por el programador, (2) el compilador analiza la estructura del código y crea un meta-modelo del mismo en memoria, (3) el compilador crea a partir de su meta-modelo la información necesaria para su acceso a través de la Mirror API por parte de los procesadores de anotaciones.

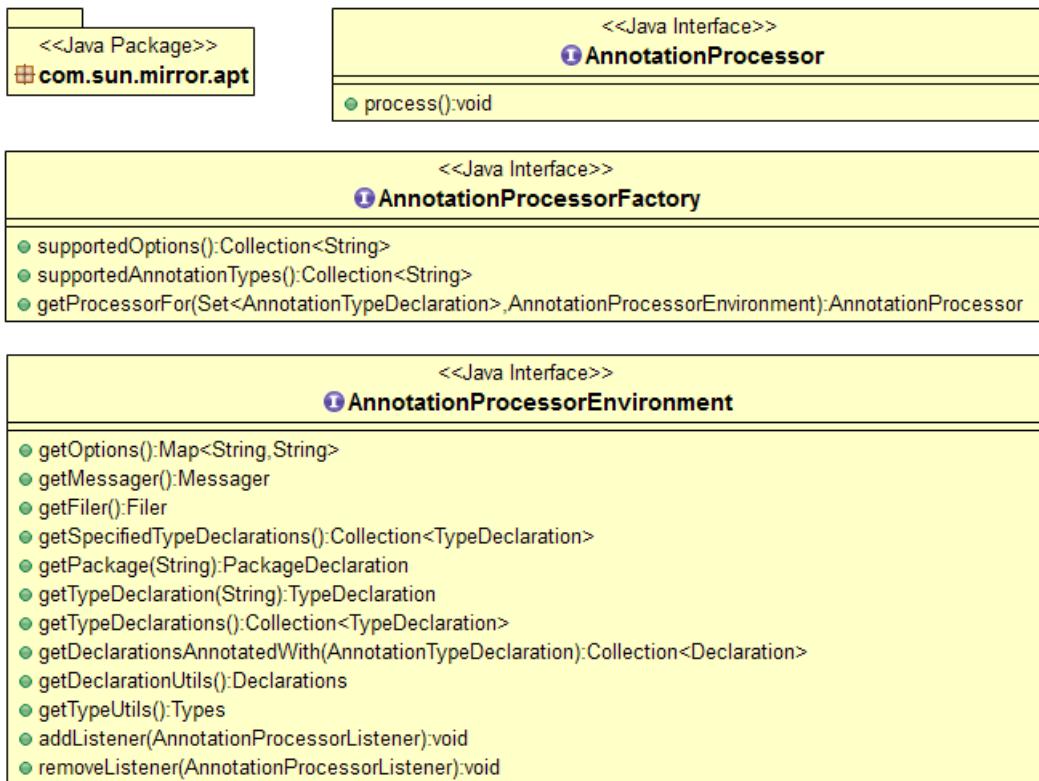
La Mirror API por tanto modela la estructura semántica de un programa escrito en Java tomando como referencia los fragmentos de código fuente de las entidades de dicho programa. Proporciona clases que representan todas las entidades que se pueden declarar en un programa Java, tales como paquetes, clases, métodos, campos, etcétera.

IMPORTANTE: La Model API no proporciona información sub-método. Es decir, que no tiene la capacidad de “ver” y, por tanto, de proporcionar información, sobre las entidades interiores a los cuerpos de los métodos, como podría ser el caso típico de las variables locales que sólo existen dentro del ámbito del cuerpo de un método. Esta es una restricción relacionada con la insuficiencia de información almacenada en los ficheros de clase generados por el compilador que fue solucionado en Java SE 8 gracias a la especificación JSR 308.

Debido a que la Model API no puede acceder a la información sub-método, **no es posible realizar procesamiento de anotaciones sobre variables locales usando procesadores de anotación J2SE 1.5** estándar. En este punto es donde resultaba relevante la aportación de [Mathias Ricken](#), profesor de la Rice University de Houston (Texas, EEUU), al crear [LAPT-javac](#) (Local Variable Annotation Enabled javac), como se comentó en el apartado dedicado a la historia de las anotaciones en Java. No obstante, al no ser estándar, queda más bien como una referencia de interés a nivel más teórico o académico que realmente práctico.

10.3.2.1.- Paquete com.sun.mirror.apt.

com.sun.mirror.apt – Arquitectura general de procesamiento de anotaciones	
Interfaz	Descripción
AnnotationProcessor	Procesadores de anotaciones J2SE 1.5. Sólo tiene el método process() .
AnnotationProcessorFactory	Factorías creadoras de instancias de procesadores de anotaciones.
AnnotationProcessorEnvironment	Entorno con información de estado necesaria para los procesadores.

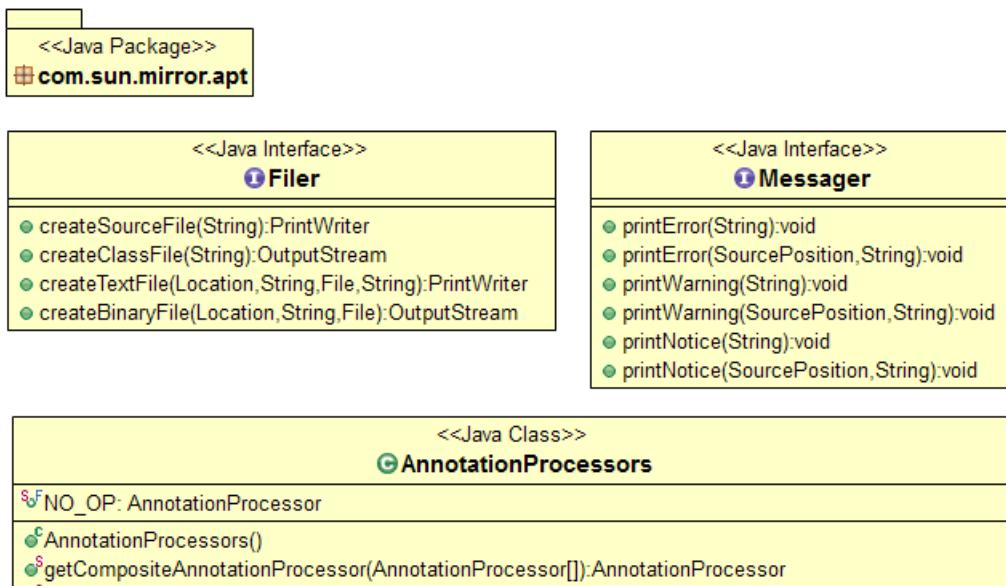


El paquete com.sun.mirror.apt es el más importante de la Mirror API. Contiene todas las interfaces sobre las que descansa el funcionamiento del procesamiento de anotaciones. La interfaz **AnnotationProcessor** es la que tiene que implementar una clase para poder considerarse procesador de anotaciones J2SE 1.5.

AnnotationProcessorFactory es la interfaz de las clases factoría de procesadores de anotaciones, que crearán nuevas instancias de **AnnotationProcessor** cuando sean necesarias. Como ya vimos, **apt** llamará al método **getProcessorFor** de las factorías cuando requiera de procesadores en cada ronda de procesamiento. Las factorías de procesadores juegan un papel muy importante, puesto que en ellas los desarrolladores de los procesadores pueden alojar variables de estado que permanecerán a lo largo de todas las rondas de procesamiento.

AnnotationProcessorEnvironment modela el “**entorno de procesamiento**”. A efectos prácticos, es un cajón de sastre de donde se pueden obtener instancias de diversas interfaces fundamentales para llevar a cabo la labor de procesamiento (**Filer**, **Messager**, **Types** y **Declarations** sobre todo). Además, ofrece métodos adicionales de conveniencia para recuperar información importante de forma cómoda.

com.sun.mirror.apt - Facilidades auxiliares	
Interfaz / Clase	Descripción
Filer	Permite crear nuevos ficheros: de código fuente, de clase, binarios y de texto.
Messager	Permite enviar mensajes al compilador: avisos informativos, warnings y errores.
AnnotationProcessors	Clase con métodos de conveniencia para componer múltiples procesadores de anotaciones en uno siguiendo el orden especificado por una colección dada.

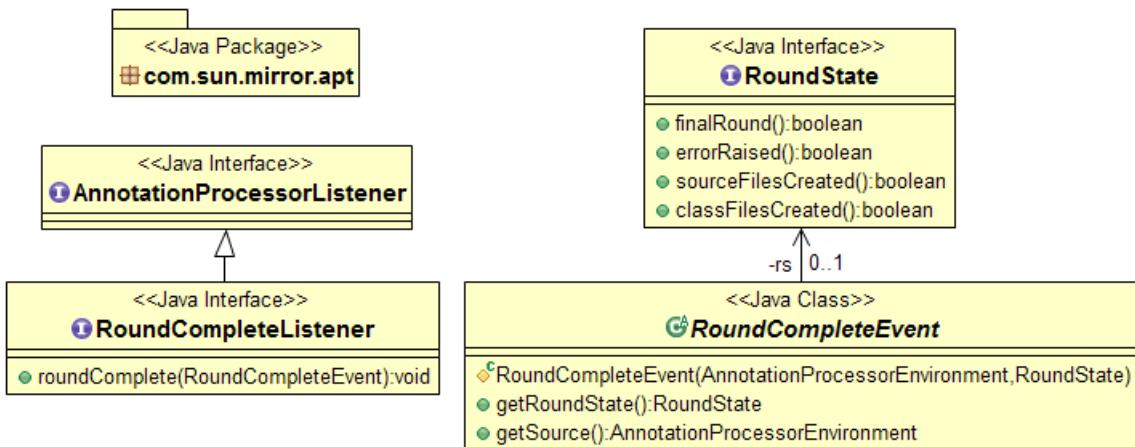


Filer es la interfaz que ofrece la Mirror API a través de la cual se implementan las facilidades de creación de nuevos ficheros durante el procesamiento de anotaciones. A través del método `getFiler()` del entorno `AnnotationProcessorEnvironment` podemos recuperar una implementación de la misma que nos permitirá crear ficheros de cuatro tipos: (1) ficheros de código fuente, (2) ficheros de clase, (3) ficheros normales de texto y (4) ficheros normales binarios. Los métodos de los ficheros de texto o binarios tienen más argumentos para poder especificar el paquete al que mapearlos (si es que así se desea) o el nombre completo del fichero. Por supuesto, **es la interfaz Filer la que habilita a los procesadores de anotaciones para la generación de nuevos ficheros**, tanto de código, como de otros tipos.

Messager es la interfaz que implementa la comunicación con el compilador. Aunque sus métodos tienen la forma `printX`, no es que impriman por consola o a fichero, si no que transmiten al compilador la información de que debe emitir un mensaje informativo, un warning o un error. La salida de dicha información deberá ser tratada de igual forma que la del compilador, ya sea en una consola de comandos o en un IDE. **Lo importante de la interfaz Messager es que al permitir emitir errores de compilación, esto habilita a los procesadores de anotación para poder implementar funciones de validación de todo tipo**, ya que, si emiten un error impiden la compilación del código al completo.

La clase **AnnotationProcessors** es simplemente una clase que implementa métodos para la composición de varios procesadores como si fueran uno solo en el orden dado por la colección con la que se pasen a su método `getCompositeAnnotationProcessor`. También contiene como campo estático de clase `AnnotationProcessors.NO_OP`, que es lo que podríamos llamar “procesador de anotaciones identidad”, ya que no realiza ninguna operación (de ahí su nombre `NO_OP`) y que no alberga ningún estado interno. Apenas se usa.

com.sun.mirror.apt - Listeners	
Interfaz / Clase	Descripción
<code>AnnotationProcessorListener</code>	Superinterfaz marcadora (sin métodos) para los eventos de los procesadores.
<code>RoundCompleteListener</code>	Evento de ronda completa. Recibe info en un objeto <code>RoundCompleteEvent</code> .
<code>RoundCompleteEvent</code>	Clase que contiene toda la información del evento <code>roundComplete</code> .
<code>RoundState</code>	Interfaz que proporciona información básica sobre el estado de una ronda.



La Mirror API ofrece la posibilidad de que los procesadores de anotaciones registren listeners para el evento `roundComplete` añadiendo una clase que implemente la interfaz `RoundCompleteListener` a través de `AnnotationProcessorEnvironment.addListener`. De este modo, los procesadores de anotaciones pueden llevar a cabo cualquier tipo de acción relacionada con su lógica y que esté vinculada de algún modo a las diferentes iteraciones dadas por las rondas de procesamiento.

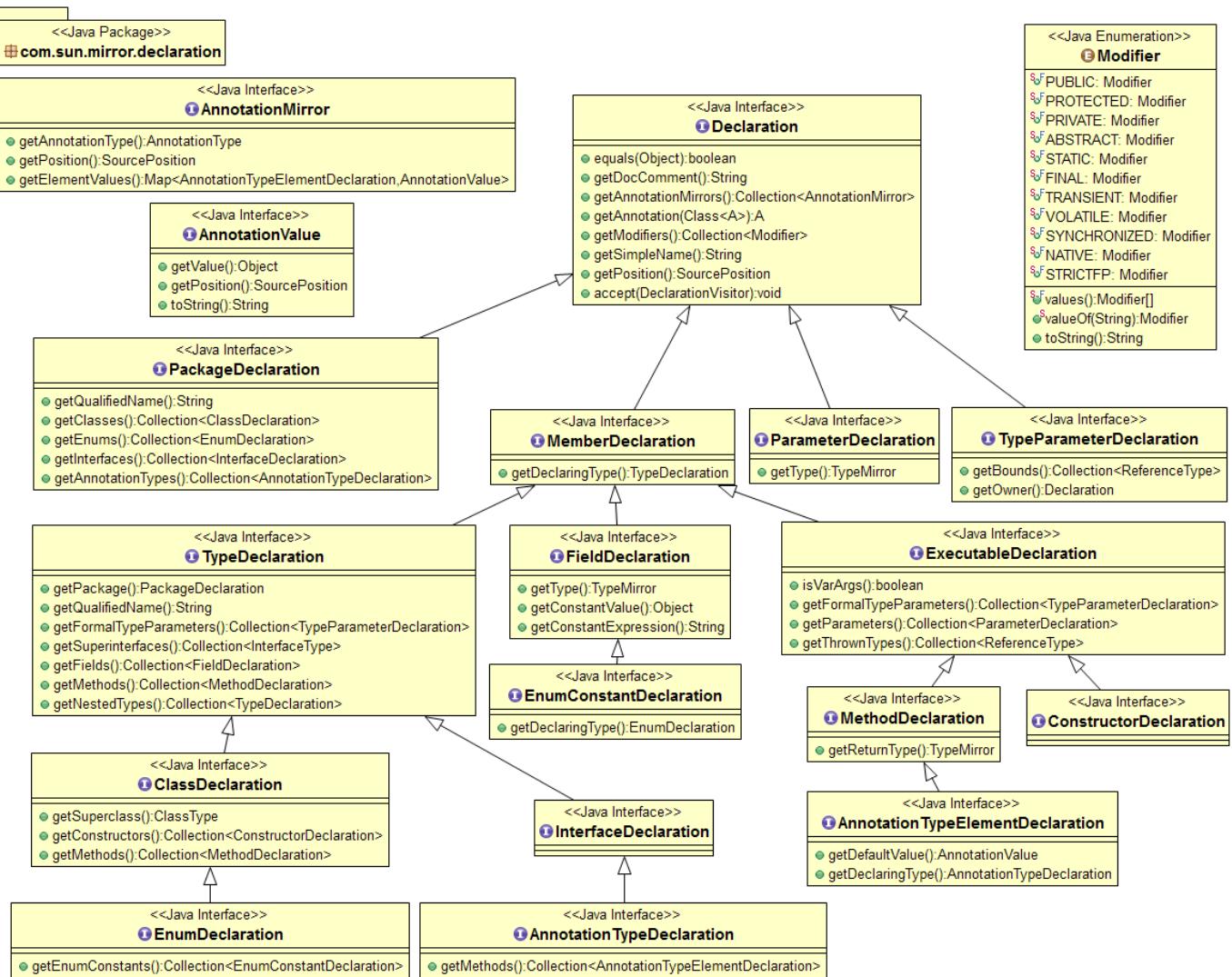
Viendo los métodos de `RoundCompleteEvent`, asociada al evento `roundComplete`, en el momento de que se complete una ronda, podemos comprobar que los procesadores reciben una completa información sobre el estado de la ronda completada gracias a los objetos `RoundState`, así como a la referencia del entorno `AnnotationProcessorEnvironment`.

La interfaz `RoundState`, que sólo se utiliza en `RoundCompleteEvent`, ofrece una muy completa información sobre el estado de la ronda recién completada: si dicha ronda es la ronda final, si arrojó algún error, o si se generaron ficheros de código fuente o de clase.

NOTA: Todo esta parte relacionada con la implementación de los listeners no sobrevivió con el paso de J2SE 1.5 a la especificación JSR 269, donde no hay ningún listener para los procesadores de anotaciones. Esto se debe a que en el nuevo diseño de la API de procesamiento de anotaciones dada por JSR 269 (paquete `javax.annotation.processing`) la información que proporciona aquí la interfaz `RoundState` está directamente disponible para los procesadores en todo momento, careciendo así de sentido alguno implementar un listener para poder recibirla.

10.3.2.2.- Paquete com.sun.mirror.declaration.

com.sun.mirror.declaration – Declaraciones	
Interfaz / Clase	Descripción
Declaration y sus subinterfaces	Interfaces que modelan las declaraciones de elementos del lenguaje Java.
AnnotationMirror	Representa la declaración de una anotación y guarda sus valores.
AnnotationValue	Valor de un elemento dado de una anotación.
Modifier	Clase enumerada que modela todos los modificadores de una declaración.

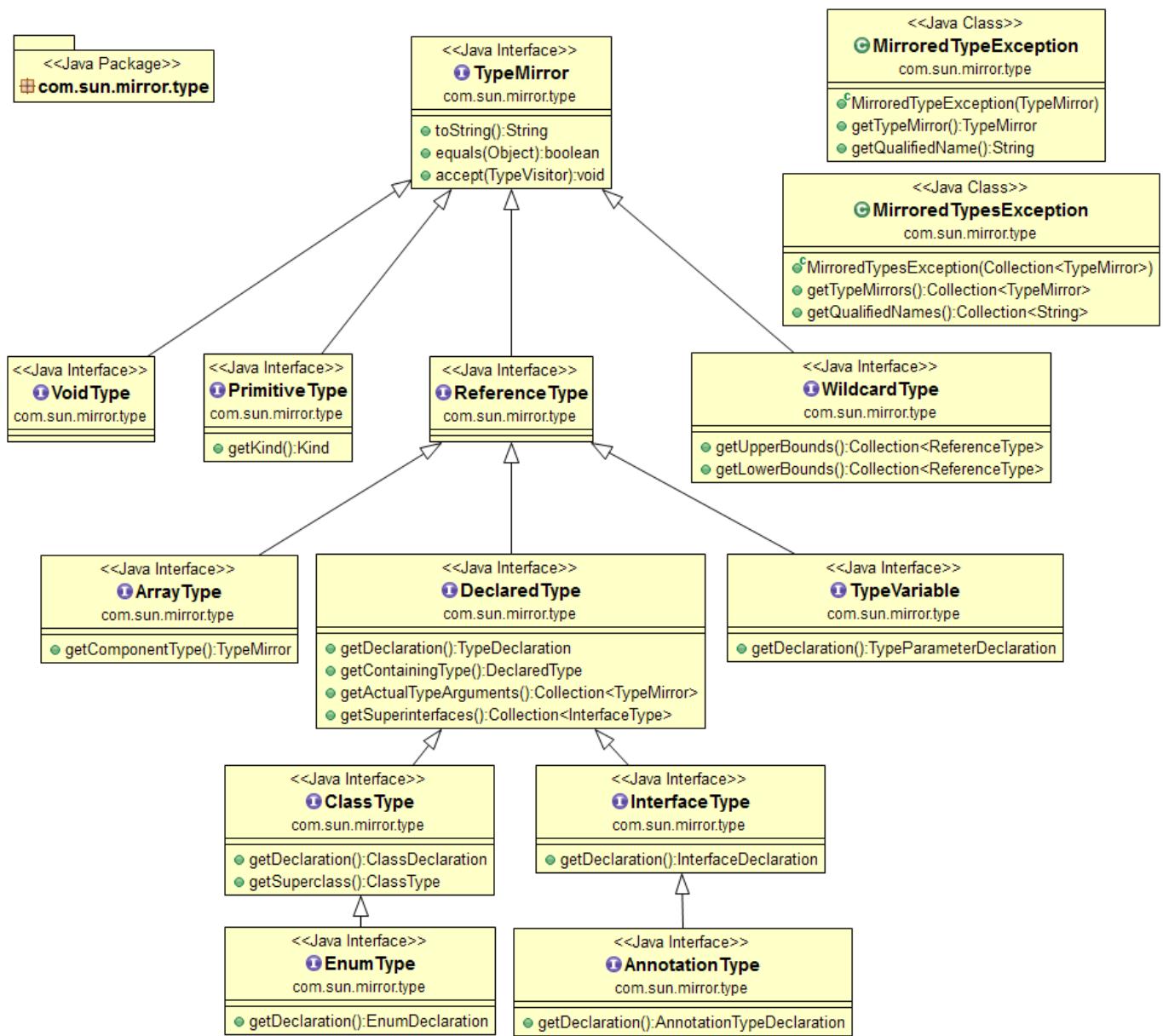


La interfaz **Declaration** es la base de la jerarquía de todas las declaraciones de elementos de programa Java. Nótese como **TypeDeclaration** agrupa a clases e interfaces, y cómo los enumerados son una especialización de las clases. Por su parte, los tipos anotación son una especialización de las interfaces. **MemberDeclaration** también es muy importante, ya que agrupa a todos aquellos elementos que pueden estar contenidos dentro de otros.

AnnotationMirror es vital, ya que será la interfaz de la que extraeremos información sobre los valores de los elementos de las anotaciones concretas del código fuente a procesar. **CUIDADO:** `getElementValues` sólo devuelve los valores de los elementos presentes en el código. Para los no presentes, habrá que recuperar sus valores por defecto por otra vía.

10.3.2.3.- Paquete com.sun.mirror.type.

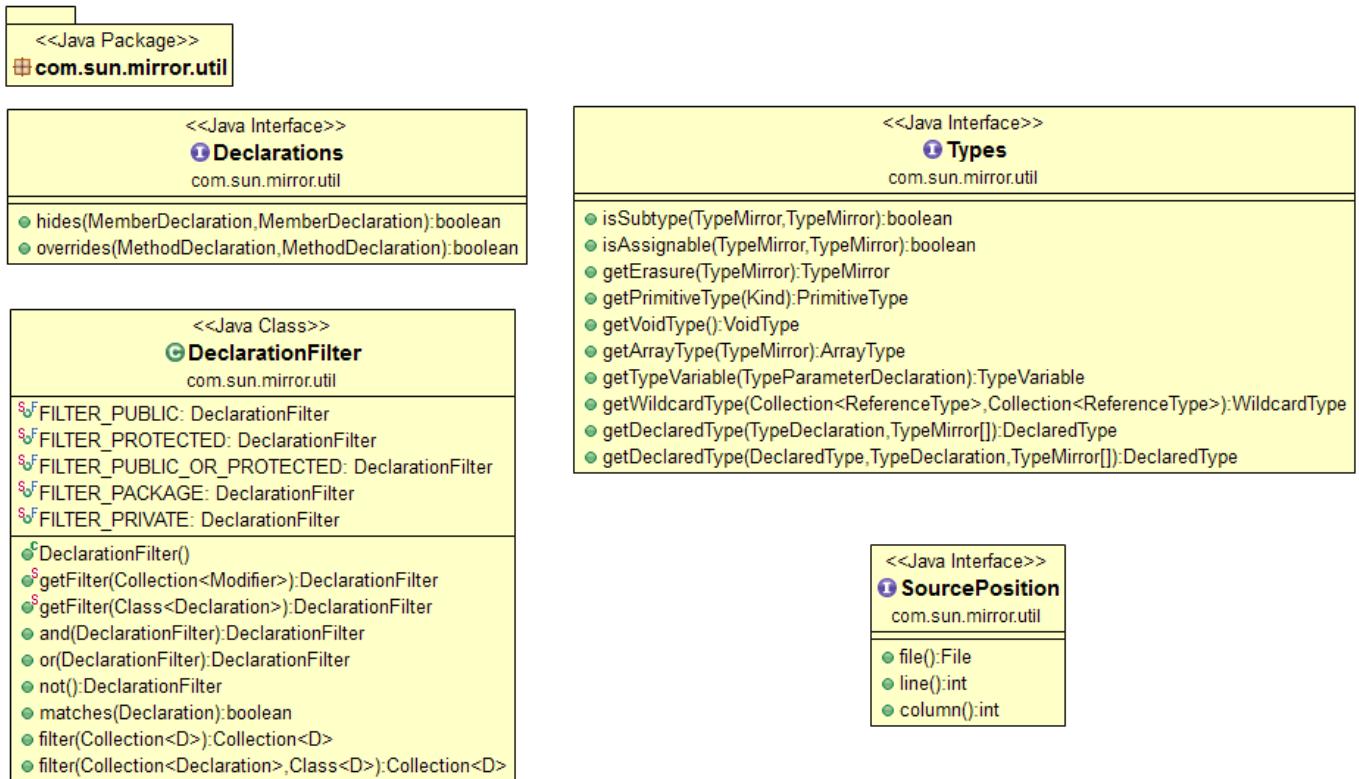
com.sun.mirror.type – Tipos	
Interfaz / Clase	Descripción
TypeMirror y sus subinterfaces	Interfaces que modelan los distintos tipos del lenguaje Java.
MirroredType[s]Exception	Excepciones producidas al intentar acceder a la Class de un TypeMirror.



La interfaz **TypeMirror** es la base de toda la jerarquía de todos los tipos Java, como **Declaration** de las declaraciones. Existe un cierto paralelismo entre ciertas relaciones que también se daban en las declaraciones, como que la interfaz **DeclaredType** agrupe de nuevo a clases e interfaces, y cómo los enumerados son una especialización de las clases, y los tipos anotación una especialización de las interfaces. **ReferenceType** es una interfaz importante al agrupar todos los tipos por referencia. **TypeVariable** modela la genericidad y **wildcardType** los comodines (?) y sus límites (“bounds”) superior (? extends T) e inferior (? super T).

10.3.2.4.- Paquete com.sun.mirror.util.

com.sun.mirror.util – Utilidades de procesamiento	
Interfaz / Clase	Descripción
Declarations	Operaciones sobre declaraciones. Sólo contiene los métodos hide y overrides .
DeclarationFilter	Permite obtener filtros sobre declaraciones, directamente o redefiniendo matches .
Types	Operaciones sobre tipos para obtener información desde declaraciones u otros tipos.
SourcePosition	Posición (fichero, línea y columna) de un fragmento de código fuente de una declaración.

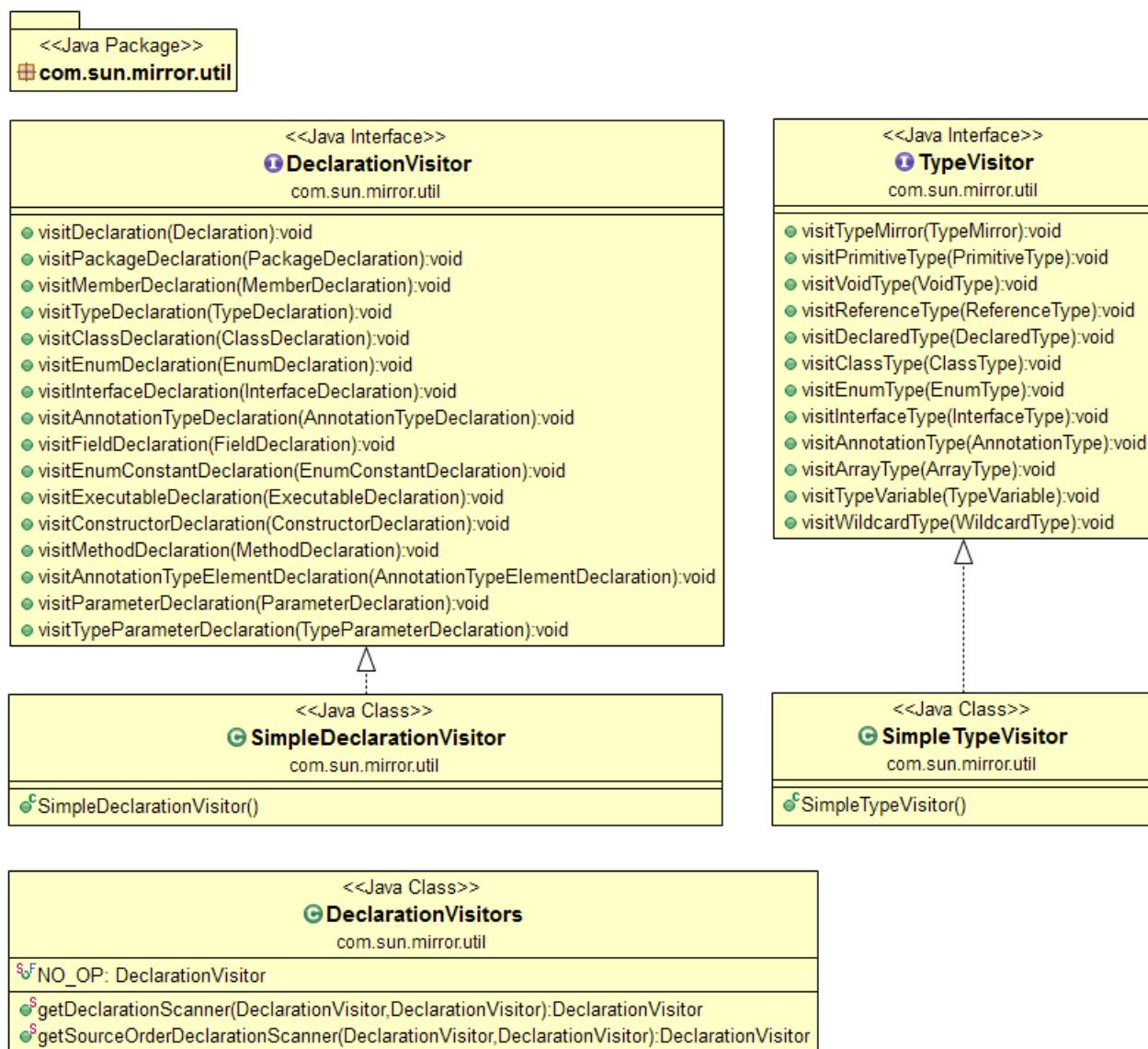


Declarations contiene los métodos sobre declaraciones para saber si una declaración oculta o redefine a otra. Ninguna de estas operaciones suele utilizarse con frecuencia.

DeclarationFilter permite crear filtros directamente a través de sus métodos estáticos **getFilter** pasándoles una colección de modificadores o una declaración. Además, ofrece métodos de composición (**and**, **or** y **not**), así como la posibilidad de implementar filtros más específicos heredando y redefiniendo el método **matches**. **DeclarationFilter** es muy importante porque una característica de Mirror API es que se manejan listas de muchos objetos y se hace bastante engorroso tener que estar continuamente iterando sobre ellas sólo para quedarnos con los objetos del tipo que nos interesa.

Types agrupa un montón de operaciones sobre los tipos que permitirán a los procesadores extraer instancias de tipos a partir de declaraciones o incluso otros tipos. Lo más común será utilizar estos métodos para obtener un **Type** a partir de alguna instancia de **Declaration** usando **getDeclaredType** (nótese este método se puede invocar con un único argumento pasándole una **TypeDeclaration**). También permite obtener instancias de tipos que no se corresponden con ninguna declaración, como ocurre con el tipo **VoidType**.

com.sun.mirror.util – Facilidades para implementar el patrón Visitor	
Interfaz / Clase	Descripción
<code>DeclarationVisitor</code>	Interfaz que define operaciones para visitar todas las declaraciones existentes.
<code>SimpleDeclarationVisitor</code>	Implementa <code>DeclarationVisitor</code> . No modifica, pero sí delega entre llamadas.
<code>DeclarationVisitors</code>	Factoría que permite crear visitantes de declaraciones especializados.
<code>TypeVisitor</code>	Interfaz que define operaciones para visitar todos los tipos existentes.
<code>SimpleTypeVisitor</code>	Implementa <code>TypeVisitor</code> . No modifica nada, pero sí delega entre llamadas.



El patrón Visitor está especialmente indicado para moverse por grandes árboles de objetos que tengan pocos tipos diferentes, como ocurre con los árboles de objetos que se acaban recorriendo en la Mirror API. Definiendo un Visitor se puede implementar de forma cómoda y sencilla una lógica sobre todos o una parte de las declaraciones y tipos procesados.

`DeclarationVisitor` y `TypeVisitor` son las interfaces para visitar declaraciones y tipos. Ambas disponen de implementaciones sencillas que no hacen nada si no delegan llamadas hacia arriba según las relaciones de herencia, por ejemplo: al visitar un tipo anotación con `visitAnnotationTypeDeclaration`, por coherencia, este invoca a `visitInterfaceDeclaration`, ya que, como hemos explicado, un tipo anotación ES UNA (especialización de) interfaz.

10.3.2.5.- Declaraciones vs Tipos.

Antes de entrar a escribir código usando las clases de la Mirror API es necesario ponernos en guardia sobre dos conceptos distintos que son muy importantes tener claros, y cuya comprensión es fundamental: la **distinción entre Declaraciones y Tipos**. De hecho, dado que se vio que esto podía provocar mucha confusión, en la documentación oficial de la API se hace mucho hincapié en tratar de explicar esta separación de conceptos.

Las Declaraciones (paquete `com.sun.mirror.declaration`), se puede decir que **son abstracciones de fragmentos del código fuente que declara (de ahí su nombre)** un elemento de código Java. Es muy importante que tengamos claro que una declaración siempre representa los elementos de programa Java dados por un determinado fragmento del código fuente. Es decir, que **las declaraciones se corresponden uno-a-uno con fragmentos de código fuente**. Por ejemplo: si declaramos una clase `public class Clase<T>`, al ser la declaración de una clase, esto se representará con un objeto `ClassDeclaration`.

Los Tipos (paquete `com.sun.mirror.type`) se puede decir que **son abstracciones de los tipos de datos Java una vez resueltos por el compilador**. La necesidad de incluir este tipo de interfaces se debe a que sirven para modelar la genericidad, también incorporada al lenguaje en JS2SE 1.5, al mismo tiempo que las anotaciones y su procesamiento.

Correspondencia Declaración en código fuente ↔ Tipos posibles		
	Declaración de tipo en código fuente (parámetros formales)	Tipos posibles (parámetros reales)
Sin genericidad	<code>java.util.List</code>	<code>java.util.List</code>
Con genericidad	<code>java.util.List<T></code>	<code>java.util.List<T> (IMPOSIBLE)</code> <code>java.util.List<String></code> <code>java.util.List<Integer></code> ... <code>java.util.List</code>
	<code>java.util.List<String></code>	<code>java.util.List<String></code>
	<code>java.util.List<Integer></code>	<code>java.util.List<Integer></code>

	<code>java.util.List</code>	<code>java.util.List</code>

Si volvemos a la declaración `public class Clase<T>`, era de tipo `Clase<T>`, ya que las declaraciones siempre tienen como tipo el que se haya escrito directamente en el código fuente. Mientras tanto, su objeto de tipo, que en este caso sería un `ClassType`, podría corresponder tanto a `Clase<String>`, como a `Clase<Integer>`, o incluso simplemente a `Clase` (un “raw type”, traducible como “tipo basto”, “tipo raso” o “tipo simple”). Así pues, una declaración puede producir múltiples tipos si tiene un parámetro de tipo `T`. No obstante, una declaración sin tipo parámetro como `Clase<Boolean>` sólo producirá el mismo tipo `Clase<Boolean>`.

La clave para diferenciar entre declaración y tipo es darse cuenta de que **un tipo nunca puede estar indeterminado**. Es decir: una declaración puede ser `Clase<T>` si así se declara en el código fuente, pero un tipo no. Nótese como en la tabla ilustrativa el tipo genérico no concretado `java.util.List<T>` aparece **tachado**. Esto es así para indicar claramente que eso es imposible. Y es que, cuando el código es analizado por el compilador, `T` será asignado a un tipo concreto, o de la estructura del código se deberá poder inferir que así será en su ejecución.

Debido a lo anterior, a los parámetros de tipo de instancias de declaraciones se les conoce como “**parámetros formales**” (“**formal parameters**”) y a los parámetros de tipo de las instancias de tipos se les conoce como “**parámetros reales**” (“**actual parameters**”).

Los “**parámetros formales**” son los **parámetros de tipo directamente escritos en el código fuente por el programador**. Deben su nombre de “formales” al hecho de que sirven para realizar una especificación de la forma que tiene que tener un tipo sin llegar a concretarlo necesariamente. En Java, los parámetros formales se implementan a través de los parámetros genéricos de tipo como `<T>`, así como los tipos comodín (“wildcard”), que no son más que parámetros formales a los que se les añade la restricción de unos límites (“bounds”) superior (`<? extends T>`) e inferior (`<? super T>`).

Los “**parámetros reales**” son **los que se resuelven utilizar realmente** cuando se ejecuta el código (de ahí lo de “reales”). En Java, los parámetros reales son inferidos por el compilador mediante sus reglas de resolución o “inferencia de tipos”, de tal manera que de una declaración con parámetros formales siempre se puedan extraer los parámetros reales.

Los parámetros formales corresponden a los datos en las declaraciones, y no en todos los tipos de declaraciones, ya que **hay declaraciones a las que no les corresponde ningún tipo**, como por ejemplo la declaración de un paquete, a la que no tendrá sentido tratar de asignarle un tipo. En la siguiente tabla se explica como obtener el tipo simple (sin parámetros de tipo) de una declaración. Para recuperar un tipo genérico específico pasarle los tipos detrás como argumento al método `Types.getDeclaredType(decl, tipo1, tipo2, ...)`.

NOTA: Las declaraciones de métodos, constructores y elementos de los tipos anotación no se consideran declaraciones de un tipo en sí mismas (no existe un `MethodType`). No obstante, como sí pueden dar lugar a un tipo (su tipo de retorno), para dichos casos, se describe como obtener el tipo de retorno de los mismos (para los constructores se ha tomado el tipo que construyen, es decir, el de su clase declarante).

Método habitual para recuperar el tipo simple (sin parámetros de tipo) de una declaración	
Declaración	Método habitual de recuperar el tipo (partiendo del tipo declaración correspondiente)
<code>AnnotationMirror</code>	<code>AnnotationType annotationType = annotMirror.getAnnotationType();</code>
<code>AnnotationTypeDeclaration</code>	<code>AnnotationType annotationType = (AnnotationType) typeUtils.getDeclaredType(annotTypeDecl);</code>
<code>AnnotationTypeElementDeclaration</code>	<code>TypeMirror typeMirror = annotTypeElementDecl.getReturnType();</code>
<code>ClassDeclaration</code>	<code>ClassType classType = (ClassType) typeUtils.getDeclaredType(classDecl);</code>
<code>ConstructorDeclaration</code>	<code>ClassType tipoConstruidoClassType = (ClassType) typeUtils.getDeclaredType(ctrDecl.getDeclaringType());</code>
<code>Declaration</code>	No procede. La interfaz <code>Declaration</code> es demasiado general.
<code>EnumConstantDeclaration</code>	<code>TypeMirror tipoConstanteEnumTypeMirror = enumConstantDecl.getType();</code>
<code>EnumDeclaration</code>	<code>EnumType enumType = (EnumType) typeUtils.getDeclaredType(enumDecl);</code>
<code>ExecutableDeclaration</code>	No procede. La interfaz <code>ExecutableDeclaration</code> es demasiado general.
<code>FieldDeclaration</code>	<code>TypeMirror tipoCampoTypeMirror = fieldDecl.getType();</code>
<code>InterfaceDeclaration</code>	<code>InterfaceType interfaceType = (InterfaceType) typeUtils.getDeclaredType(interfaceDecl);</code>
<code>MemberDeclaration</code>	No procede. La interfaz <code>MemberDeclaration</code> es demasiado general.
<code>MethodDeclaration</code>	<code>TypeMirror tipoRetornoTypeMirror = metodoDecl.getReturnType();</code>
<code>PackageDeclaration</code>	No procede. No tiene sentido aplicar la noción de tipo a un paquete.
<code>ParameterDeclaration</code>	<code>TypeMirror tipoParamTypeMirror = paramDecl.getType();</code>
<code>TypeDeclaration</code>	<code>DeclaredType declaredType = typeUtils.getDeclaredType(typeDecl);</code>
<code>TypeParameterDeclaration</code>	No procede. No tiene sentido. Precisamente los parámetros de tipo <code><T></code> son los que desaparecen cuando se determina un tipo. Son los que menos podrían tener un tipo.

Ejemplo: Diferencias entre Declaración y Tipos en las definiciones de clase

Imaginemos que definimos las clases siguientes:

```
public class ClaseA<T>
public class ClaseB<T extends Number> extends ClaseA<T>
public class ClaseC extends ClaseB<Integer>
```

La **ClaseA** tiene un parámetro de tipo, la **ClaseB** hereda de **ClaseA** y le coloca un límite superior a la definición del tipo, forzando a que sea o herede de la clase **Number**. Finalmente, la **ClaseC** simplemente concretiza el parámetro de tipo a **Integer** (que efectivamente hereda de **Number**) y hereda de dicha clase para crear una clase no parametrizada al final de la jerarquía. Veamos qué valores de declaraciones y tipos tenemos para cada clase. Además, para aclarar otro concepto que explicaremos a continuación incluimos también el valor del tipo de la superclase.

Correspondencia Declaración en código fuente ↔ Tipos posibles			
Clase	Declaración (parámetros formales)	Tipo (parámetros reales)	SuperTipo o Tipo de la superclase (parámetros reales)
ClaseA	ClaseA<T>	ClaseA	java.lang.Object
ClaseB	ClaseB<T extends java.lang.Number>	ClaseB	ClaseA
ClaseC	ClaseC	ClaseC	ClaseB<java.lang.Integer>

¿Cómo interpretamos estos resultados? Lo primero es ver cómo se pierde la información de los parámetros de tipo al pasar de Declaración a Tipo. ¿Pero eso quiere decir que un tipo nunca va a tener información de parámetros de tipo? No, como demuestra el tipo de la superclase de **ClaseC**, que es **ClaseB<java.lang.Integer>**.

Entonces, ¿por qué el tipo de una declaración nunca tiene parámetros de tipo en su propio tipo? Pues porque el parámetro de tipo sólo se puede concretizar en las subclases. **ClaseA** dentro de su definición nunca puede darle un valor a **T**, por lo que el compilador le asigna como tipo su tipo simple. Sólo cuando hay herencia se le puede dar un valor a **T**. En el caso de **ClaseC**, como se ve en su declaración, hereda de **ClaseB<java.lang.Integer>**, que es justo lo que la Mirror API devuelve como tipo de su superclase. En este caso, **T** ha sido concretado como **Integer**, respetando además la restricción dada por **ClaseB** de que tenía que ser un **Number**.

De este ejemplo podemos extraer una noción muy importante para poder separar y discernir correctamente los conceptos de declaración y de tipo: todas las declaraciones (siempre que declaren un tipo) tienen asociadas un objeto **Declaration** y un **TypeMirror**, pero, además, si la declaración define un parámetro de tipo variable **<T>**, cada subclase puede tener un supertipo específico. Este es el caso de **ClaseC** con **ClaseB<java.lang.Integer>**. Si creáramos una **ClaseC2** que también heredara de **ClaseB** con **public class ClaseC2 extends ClaseB<Float>**, tendríamos ese resultado del que tanto se habla en la documentación oficial de la Mirror API: una declaración **ClaseB<T extends java.lang.Number>** que no produce únicamente un tipo (antigua correspondencia uno-a-uno, cuando no había genericidad), si no múltiples tipos concretos, en este caso: **ClaseB<java.lang.Integer>** y **ClaseB<java.lang.Float>**.

10.3.2.6.- Mirrors: A través del espejo.

Otra de las particularidades que tiene la Mirror API es el uso de “Mirrors”. Aunque son muy importantes, no son muy numerosos; básicamente dos interfaces, una para modelar las anotaciones y otra para los tipos, junto con dos excepciones relacionadas:

Mirrors de la Mirror API y sus excepciones	
Nombre cualificado	Descripción
<code>com.sun.mirror.declaration.AnnotationMirror</code>	Representa la declaración de una anotación.
<code>com.sun.mirror.type.TypeMirror</code>	Base de la jerarquía de tipos de la Mirror API.
<code>com.sun.mirror.type.MirroredTypeException</code>	Excepción arrojada cuando una aplicación trata de acceder al objeto <code>Class</code> de un TypeMirror.
<code>com.sun.mirror.type.MirroredTypesException</code>	Excepción arrojada cuando se trata de acceder a una secuencia de objetos <code>Class</code> de un TypeMirror (p. ej. un array <code>Class[]</code> elemento de un tipo anotación).

Las existencia de los “Mirror” no tiene una justificación conceptual, como sí la tenía la separación de conceptos entre Declaraciones y Tipos, si no que es provocada por circunstancias digamos “subterráneas”. Para entender la necesidad de introducir la figura de los “Mirrors” por parte de los diseñadores de la Mirror API, es muy importante tener en cuenta un detalle que a veces puede pasarse por alto, pero que es trascendental: **el procesamiento de anotaciones se lleva a cabo en tiempo de compilación.**

El compilador Java no carga la definición de las clases de forma secuencial o completa, si no siguiendo complejas estrategias de optimización. Esto implica que, durante el proceso de compilación, en el entorno en el que se realiza el procesamiento de anotaciones, **no es posible disponer de cierta información de las clases normalmente almacenada en tiempo de ejecución en una instancia de `Class`**. Por este motivo, cuando los diseñadores de la Mirror API se encontraron con este problema, crearon el **concepto de los “Mirrors” o “imágenes especulares”**: algo similar a lo que sería un objeto proxy, pero parcial; a efectos prácticos, los “Mirrors” serían más bien como una **fachada segura a la que se puede pedir información** con la certeza de que no nos encontraremos con algún problema debido a encontrarnos en tiempo de compilación.

Como se introdujeron de mala gana, se crearon sólo dos “interfaces Mirror” únicamente para los objetos que podían dar problemas por no estar cargados por el compilador: la meta-information de las anotaciones (de aquí surgió `AnnotationMirror`) y la información de tipos a través de `Class` (de ahí `TypeMirror`). Debido a que la necesidad de los “Mirrors” se debe a la implementación del compilador Java, y el compilador seguía funcionando igual en Java SE 6, las “interfaces Mirror” y sus excepciones siguen presentes en la nueva API de procesamiento de anotaciones JSR 269 llamada Model API, además con los mismos nombres.

Por tanto, **deben tratar de no utilizarse las facilidades de la API de reflexión siempre que la Mirror API ofrezca una alternativa para acceder a la información que necesitamos** para realizar el procesamiento de anotaciones, ya que, en caso de tratar de acceder a una información de reflexión vía el objeto `Class` y dicha información no estar disponible por no haber sido cargada por el compilador, nos encontraremos con que en dicho punto del código el entorno de compilación arrojará una `MirroredTypeException` (o `MirroredTypesException` si se trata de acceder a un array o colección de objetos `Class`).

El ejemplo más claro de los posibles problemas que podemos tener al intentar acceder a información no disponible en tiempo de compilación lo tenemos al tratar de recuperar información

sobre las anotaciones que anotan una determinada declaración. Esto puede hacerse, desde el entorno de un procesador de anotaciones J2SE 1.5, entre otras vías usando dos métodos alternativos de la interfaz **Declaration**:

Métodos de recuperación de anotaciones de com.sun.mirror.declaration.Declaration	
Métodos	Descripción
<A extends Annotation> A getAnnotation(Class<A>annotationType)	Usa reflexión. CUIDADO .
Collection<AnnotationMirror> getAnnotationMirrors()	Usa "Mirrors". SEGURO .

Ejemplo: Procesamiento **erróneo** de un tipo anot. con un elemento Class

Supongamos que diseñamos el tipo anotación **@AnotacionElementoClase** que, como indica su nombre, tiene un elemento de tipo **Class**:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface AnotacionElementoClase {
    Class elementoClase() default java.lang.Object.class;
}
```

Si ahora escribiéramos un procesador que usara **getAnnotation** (el método que usa reflexión y, por tanto, peligroso en este contexto de tiempo de compilación), podríamos escribir el siguiente código para recuperar la declaración de la anotación e imprimir su valor:

```
// tratamos de recuperar la declaración del tipo anotación que queremos procesar
AnotacionElementoClase anotacion =
declaration.getAnnotation(AnotacionElementoClase.class);
if (anotacion != null) {
    // clase anotada con la anotación que queremos procesar -> mostramos su información
    msg.printNotice("Clase anotada: " + declaration.getQualifiedName());
    msg.printNotice("Anotación: @" + AnotacionElementoClase.class.getCanonicalName());
    msg.printNotice("Valor: " + anotacion.elementoClase());
}
```

Si ahora anotamos una clase cualquiera con **@AnotacionElementoClase** tal que así:

```
@AnotacionElementoClase(elementoClase=Integer.class)
public class ClaseAnotadaElementoClase { ... }
```

Al ejecutar nuestro procesador, ya que la información de **Integer.class** no ha sido cargada por el compilador, al tratar de acceder a ella obtendríamos la **MirroredTypeException**:

```
Note: Clase anotada: anotaciones.ejemplos.procesamiento.java5.mirror.ClaseAnotadaElementoClase
Note: Anotación: @anotaciones.ejemplos.procesamiento.java5.mirror.AnotacionElementoClase
Problem encountered during annotation processing;
see stacktrace below for more information.
com.sun.mirror.type.MirroredTypeException: Attempt to access Class object for TypeMirror java.lang.Integer
at com.sun.tools.apt.mirror.declaration.AnnotationProxyMaker$MirroredTypeExceptionProxy.generateException(Unknown Source)
at sun.reflect.annotation.AnnotationInvocationHandler.invoke(AnnotationInvocationHandler.java:75)
at $Proxy17.elementoClase(Unknown Source)
at
anotaciones.ejemplos.procesamiento.java5.mirror.AnotacionElementoClaseProcessor.process(AnotacionElementoClaseProcessor.java:56)
at com.sun.mirror.apt.AnnotationProcessors$CompositeAnnotationProcessor.process(AnnotationProcessors.java:84)
at com.sun.tools.apt.comp.Apt.main(Unknown Source)
at com.sun.tools.apt.main.AptJavaCompiler.compile(Unknown Source)
at com.sun.tools.apt.main.Main.compile(Unknown Source)
... resto de la traza de la pila...
```

En el ejemplo anterior se puede ver como hay una línea que curiosamente emplea reflexión (`AnotacionElementoClase.class.getCanonicalName()`) y funciona. Y es que, como hemos comentado, aunque el compilador no carga la información de todas las clases, sí que carga la mayoría necesaria para lograr que la compilación habitual no falle. Los casos en los que se sabe que esto da problemas son excepcionales y las “interfaces Mirror” son sólo para dichos casos.

Ejemplo: Procesamiento correcto de un tipo anot. con un elemento Class

Si cambiamos el código de nuestro procesador de anotaciones para usar el método `getAnnotationMirrors()`, que no usa reflexión y, por tanto, es seguro en el entorno del tiempo de compilación, nos queda un código más farragoso, ya que ahora no usaremos directamente una instancia del tipo anotación accediendo directamente a un método que nos devuelve el valor del elemento (`anotacion.elementoClase()`), si no que tendremos “Mirrors” con colecciones sobre las que iterar, primero para encontrar la anotación entre una colección de `AnnotationMirror`, y luego para encontrar el valor del elemento que buscamos entre la lista de valores de la propia anotación, por lo que el compacto código original se convierte en lo siguiente:

```
// tratamos de recuperar la declaración del tipo anotación que queremos procesar
Collection<AnnotationMirror> annotationMirrors = declaration.getAnnotationMirrors();

// buscamos la anotación de nuestro tipo @AnotacionElementoClase
for (AnnotationMirror annotationMirror : annotationMirrors) {
    if (AnotacionElementoClase.class.getCanonicalName()
        .equals(annotationMirror.getAnnotationType().toString())) {

        // anotación del tipo que buscamos - recuperamos el valor de elementoClase
        Map<AnnotationTypeElementDeclaration, AnnotationValue> mapValores = annotationMirror.getElementValues();
        for (Map.Entry<AnnotationTypeElementDeclaration, AnnotationValue> entrada :
            mapValores.entrySet()) {
            if ("elementoClase".equals(entrada.getKey().getSimpleName().toString())) {

                // clave que buscamos -> recuperamos su valor de clase
                // NOTA 1: nótese que el valor NO viene en un Class, si no en un TypeMirror
                // NOTA 2: el primer getValue() recupera un objeto AnnotationValue y
                // el segundo getValue() el Object que realmente es el valor del elemento buscado
                TypeMirror elementoClase = (TypeMirror) entrada.getValue().getValue();

                // hemos encontrado la anotación y su elementoClase -> mostramos su información
                msg.printNotice("Clase anotada: " + declaration.getQualifiedName());
                msg.printNotice("Anotación: @" + AnotacionElementoClase.class.getCanonicalName());
                msg.printNotice("Valor: " + elementoClase);
            }
        }
    }
}
```

No obstante, podría conseguirse un código igual de compacto al original si refactorizamos esta búsqueda tan engorrosa a través de un método auxiliar. Y lo mejor de todo es que usando `getAnnotationMirrors()` el procesador no corre peligro de arrojar la `MirroredTypeException`, por lo que obtenemos la salida correcta deseada en un principio:

```
Note: Clase anotada: anotaciones.ejemplos.procesamiento.java5.mirror.ClaseAnotadaElementoClase
Note: Anotación: @anotaciones.ejemplos.procesamiento.java5.mirror.AnotacionElementoClase
Note: Valor: java.lang.Integer
```

getAnnotation: Un método especial.

El primer error obtenido usando `getAnnotation`, no obstante, no debería habernos sorprendido en absoluto, ya que sí está reflejado en la documentación oficial. En la página del [javadoc oficial](#) de la interfaz `Declaration`, para el método `getAnnotation` tenemos la siguiente información (traducido del original inglés al castellano):

```
<A extends Annotation> A getAnnotation(Class<A> annotationType)
```

Devuelve la anotación de esta declaración que contiene el tipo especificado. La anotación podría ser heredada o directamente presente en esta declaración.

La anotación retornada por este método podría contener un elemento cuyo valor de tipo es `Class`. Este valor no puede ser retornado directamente: la información necesaria para localizar y cargar una clase (como por ejemplo el *class loader* a utilizar) no está disponible, y la clase podría no ser localizable en absoluto. Intentar leer un objeto `Class` invocando el método pertinente sobre la anotación retornada resultará en una `MirroredTypeException`, de la cual podría extraerse el correspondiente `TypeMirror`. De forma similar, intentar leer un elemento de tipo `Class[]` resultará en una `MirroredTypesException`.

Así pues, los desarrolladores estaban sobre aviso. Sin embargo, en la documentación de este método hay una nota especial (algo muy poco habitual en la documentación de las APIs oficiales Java) que supone la escueta explicación de la documentación oficial al motivo de incluir las “interfaces Mirror” desde un principio, además de poner de manifiesto que el método `getAnnotation` supone una excepción al usar reflexión:

Nota: Este método es diferente de los demás en esta y las interfaces relacionadas. Opera sobre información reflexiva en tiempo de ejecución -- representaciones de tipos anotación actualmente cargadas por la MV -- en lugar de sobre las representaciones especulares definidas y usadas a través de estas interfaces. Este método está dirigido a códigos cliente escritos para operar sobre un conjunto fijo y conocido de tipos anotación.

Es decir, que `getAnnotation` fue añadido a la interfaz `Declaration` para que lo usaran los desarrolladores que comprendieran todo lo que hemos explicado en este apartado y que, sabiendo que los tipos anotación (el aludido “conjunto fijo y conocido de tipos anotación”) que iban a procesar sus procesadores no tenían elementos de tipos `Class` o `Class[]` y que no corrían peligro de toparse con la `MirroredTypeException`, quisieran aprovecharse de la enorme comodidad que ofrece este método a la hora de escribir código.

En la documentación anterior se menciona que “intentar leer un objeto `Class` invocando el método pertinente sobre la anotación retornada resultará en una `MirroredTypeException`, de la cual podría extraerse el correspondiente `TypeMirror`”, ¿a qué se refiere con esto? Pues que si se intenta leer la información de clase de `Integer.class`, como en los ejemplos anteriores, dicha información sabemos que no existe, pero sí existirá el `TypeMirror` correspondiente con el valor `java.lang.Integer`, que es lo que llegamos a recuperar finalmente en el ejemplo que usa `getAnnotationMirrors()`. Por eso es que en la `MirroredTypeException` el entorno de compilación `iijns` puede ofrecer el `TypeMirror` que necesitamos!!! Veamos como funciona esto con el ejemplo de una alternativa más para obtener el valor del elemento `Class` de un tipo anotación.

Ejemplo: Procesamiento correcto alternativo de un tipo anotación con un elemento Class

Volviendo al código inicial que daba error, si capturamos la `MirroredTypeException` y accedemos al `TypeMirror` que nos devuelve, podemos obtener el valor que buscábamos, aunque en lugar de llegarnos como un objeto `Class` nos llega la información del valor del elemento `Class` como un objeto `TypeMirror`. El nuevo código alternativo podría quedar así:

```
// tratamos de recuperar la declaración del tipo anotación que queremos procesar
AnotacionElementoClase anotacion =
declaration.getAnnotation(AnotacionElementoClase.class);

if (anotacion != null) {
    // clase anotada con la anotación que queremos procesar -> mostramos su información
    msg.printNotice("Clase anotada: " + declaration.getQualifiedName());
    msg.printNotice("Anotación: @" + AnotacionElementoClase.class.getCanonicalName());
    try {
        msg.printNotice("Valor: " + anotacion.elementoClase());
    } catch (MirroredTypeException mtex) {
        msg.printNotice("Valor: " + mtex.getTypeMirror());
    }
}
```

Nótese cómo se capture la excepción y se utiliza el `TypeMirror` devuelto por la expresión `mtex.getTypeMirror()`. Este método alternativo de recuperar el valor `Class` de un tipo anotación también funciona correctamente, y nos da la respuesta correcta que buscamos:

```
Note: Clase anotada: anotaciones.ejemplos.procesamiento.java5.mirror.ClaseAnotadaElementoClase
Note: Anotación: @anotaciones.ejemplos.procesamiento.java5.mirror.AnotacionElementoClase
Note: Valor: java.lang.Integer
```

No obstante, si bien es un método que aparentemente debería funcionar en todo caso, no se considera una buena práctica de programación utilizar la captura de excepciones de forma sistemática para implementar la lógica de negocio. Por tanto, consideramos esta estrategia de procesamiento, aunque correcta, como alternativa y no recomendable.

Finalmente, comentar que esta misma estrategia es posible, como se comenta en la documentación oficial, también para arrays `Class[]`, pero en lugar de capturar la excepción `MirroredTypeException`, se capture y se recupera la colección de `TypeMirror` de cada una de las clases. Realizando un cambio rápido en el diseño de nuestra anotación para que contenga un array `Class[]` y cambiándole el nombre a `@AnotacionElementosClase`, con un código muy similar al anterior, pero capturando `MirroredTypesException`, podemos obtener por ejemplo para `@AnotacionElementosClase(elementosClase={Integer.class, Long.class})` la salida correcta esperada:

```
Note: Clase anotada: anotaciones.ejemplos.procesamiento.java5.mirror.ClaseAnotadaElementosClase
Note: Anotación: @anotaciones.ejemplos.procesamiento.java5.mirror.AnotacionElementosClase
Note: Valor: [java.lang.Integer, java.lang.Long]
```

Si implementamos esto mismo usando `getAnnotationMirrors()`, que es lo preferible, hay que comentar que como valor del tipo elemento `Class[]` en lugar de un `AnnotationValue` con un `TypeMirror` como valor, recibimos inicialmente una `AnnotationValue` con un `ArrayList<AnnotationValue>`. Los valores de las `AnnotationValue` de dicha lista no son `TypeMirror`, si no, en este caso, `ClassType`. Este resultado es bastante curioso y difiere de lo esperado en un principio, pero igualmente nos proporciona la información que necesitamos.

10.3.2.7.- Filtrado de declaraciones.

En la Mirror API trabajar con colecciones de declaraciones de diverso tipo está a la orden del día. Si además tenemos que quedarnos con un subconjunto de declaraciones que cumplan unas determinadas características, eso nos obligaría a tener que construir una lógica en un bucle **for** para cada operación de filtrado, lo que supondría todo un engorro y una auténtica pesadez.

Los diseñadores de la Mirror API identificaron el problema y, para proporcionar una forma cómoda de extraer de las colecciones de declaraciones sólo aquellas que cumplieran ciertos criterios, introdujeron los filtros de declaraciones con la clase **DeclarationFilter**.

DeclarationFilter permite crear filtros concretos directamente a través de sus constantes y métodos estáticos **getFilter** pasándoles una colección de modificadores o una clase de un tipo de declaración. Además, ofrece métodos para la composición de filtros (**and**, **or** y **not**), o implementar filtros más específicos heredando y redefiniendo el método **matches** del filtro.

<<Java Class>>	
DeclarationFilter	
com.sun.mirror.util	
\$F	FILTER_PUBLIC: DeclarationFilter
\$F	FILTER_PROTECTED: DeclarationFilter
\$F	FILTER_PUBLIC_OR_PROTECTED: DeclarationFilter
\$F	FILTER_PACKAGE: DeclarationFilter
\$F	FILTER_PRIVATE: DeclarationFilter
•C	DeclarationFilter()
•S	getFilter(Collection<Modifier>):DeclarationFilter
•S	getFilter(Class<Declaration>):DeclarationFilter
•S	and(DeclarationFilter):DeclarationFilter
•S	or(DeclarationFilter):DeclarationFilter
•S	not():DeclarationFilter
•S	matches(Declaration):boolean
•S	filter(Collection<D>):Collection<D>
•S	filter(Collection<Declaration>,Class<D>):Collection<D>

Todo lo anterior ofrece un gran abanico de posibilidades que permiten construir filtros complejos con relativamente poco esfuerzo y, dado que todas las operaciones que crean un filtro se pueden encadenar, puede dar lugar a una sintaxis similar a la de los lenguajes funcionales.

Ejemplo: Filtro sencillo para decl. clases **public** sin tipos enumerados

```
DeclarationFilter filtroClasesPublicSinTiposEnumerados =
    DeclarationFilter.FILTER_PUBLIC                                // public
    .and(DeclarationFilter.getFilter(ClassDeclaration.class))      // clase
    .and(DeclarationFilter.getFilter(EnumDeclaration.class).not()); // enum .not()
```

Ejemplo: Filtro complejo para decl. métodos con múltiples condiciones

Para crear un filtro de métodos setter (tipo retorno: **void**) con modificadores **public static**, que además tengan un único argumento de tipo **boolean**, podríamos hacer lo siguiente:

```
DeclarationFilter filtroMetodosSetterPublicStaticConArgumentoBoolean =
    DeclarationFilter.getFilter(Arrays.asList(Modifier.PUBLIC, Modifier STATIC)) // public static
    .and(DeclarationFilter.getFilter(MethodDeclaration.class))                  // método
    .and(new DeclarationFilter() {
        @Override
        public boolean matches(Declaration decl) {
            Types types = env.getTypeUtils();
            MethodDeclaration methodDecl = (MethodDeclaration) decl;
            return methodDecl.getSimpleName().startsWith("set") // nombre empieza con set
                && types.isSubtype(methodDecl.getReturnType(), types.getVoidType()) // void
                && methodDecl.getParameters().size() == 1 // tiene un único parámetro
                && types.isSubtype(methodDecl.getParameters().iterator().next().getType(),
                    types.getPrimitiveType(PrimitiveType.Kind.BOOLEAN)); // boolean
        }
    });
});
```

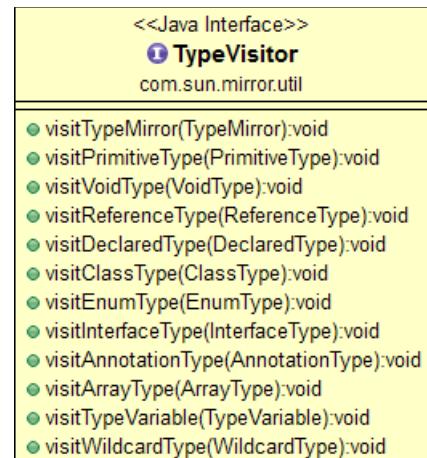
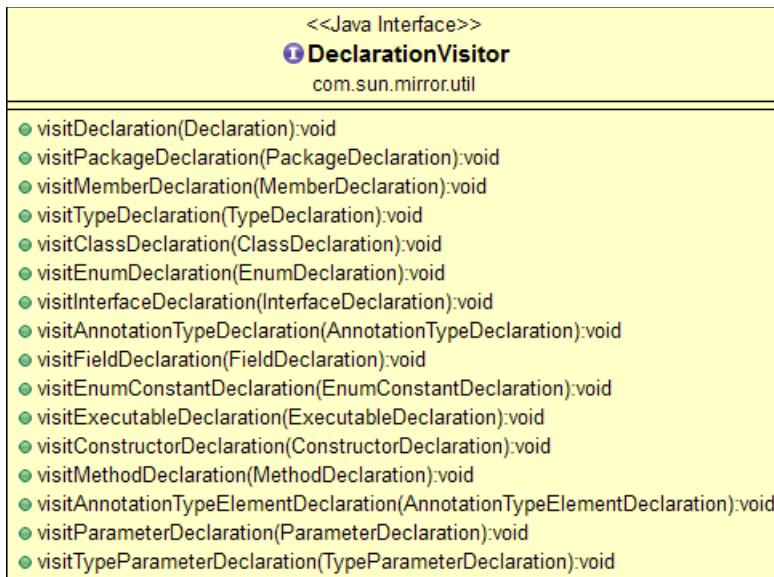
La declaración del filtro incluye un filtro anónimo, pero si tenemos filtros complejos que vamos a estar utilizando constantemente, lo mejor será definir clases filtro reutilizables.

10.3.2.8.- Usos del patrón Visitor.

En la Mirror API es muy frecuente cuando se desarrolla un procesador de anotaciones tener que implementar diversas lógicas de proceso según el tipo de elemento procesado. Por lo que se hace muy común tener construcciones condicionales de if con una condición por cada tipo de declaración o de tipo a procesar, que suelen seguir un patrón similar al siguiente:

Construcciones condicionales de Declaraciones	Construcciones condicionales de Tipos
<pre>if (decl instanceof ClassDeclaration) { // ...procesado de la declaración de clase... } else if (decl instanceof InterfaceDeclaration) { // ...procesado de la declaración de interfaz... } else if ...</pre>	<pre>if (type instanceof ClassType) { // ...procesado del tipo clase... } else if (type instanceof InterfaceType) { // ...procesado del tipo interfaz ... } else if ...</pre>

El [patrón de diseño Visitor](#) permite separar la lógica implementada de la estructura de los objetos procesados. Aunque el patrón Visitor tiene como una de sus ventajas principales el poder añadir a la jerarquía de objetos procesados nuevas operaciones sin modificarlas, esto no se cumple en la Mirror API, debido a que las interfaces Visitor de la Mirror API no definen un único método **visit** con diferentes especializaciones a las que posteriormente se le pueden añadir nuevas especializaciones al tener el método el mismo nombre, si no que en su lugar se definen métodos con distintos nombres **visitX** según el tipo de elemento a visitar.



El problema con este enfoque es que si se añadiera un nuevo tipo de elemento a modelar, esto implicaría tener que modificar la interfaz correspondiente para añadir un nuevo método y, por tanto, modificar también todas sus respectivas implementaciones. No obstante, a efectos prácticos no existiría finalmente dicho problema de diseño, ya que la Mirror API fue totalmente apartada y deprecada cuando en Java SE 6 se introdujo el procesamiento de anotaciones dado por la especificación JSR 269, donde ya sí se añadiría un método genérico **visit** a secas, pero se seguirían dejando también los métodos especializados **visitX**.

La verdadera utilidad de las clases relacionadas con el patrón Visitor de la Mirror API es proporcionar una forma cómoda y sencilla de separar la lógica de procesamiento de anotaciones de determinados tipos de elementos.

Ejemplo: Declaración de un SimpleDeclarationVisitor anónimo sobre declaraciones de clases

Supongamos que se desea un tipo anotación para una herramienta de análisis de código que permitiría contabilizar el número total de clases y métodos definidos y calcular su media. Diseñamos el siguiente tipo anotación marcadora para anotar las clases a procesar:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface AnotacionEjemploVisitor { }
```

Computar la media del número de métodos por clase para la herramienta de análisis, podría implementarse sin ningún Visitor de una forma similar a la siguiente, donde se van actualizando y mostrando las estadísticas a medida que se encuentran nuevas clases o métodos:

```
// iteramos sobre las declaraciones anotadas a ser procesadas
for (Declaration declaration : env.getDeclarationsAnnotatedWith(annotationTypeDecl)) {

    // PROCESAMIENTO DE LA DECLARACIÓN (SIN VISITOR)

    // comprobamos si la declaración es sobre una clase, ya que la anotación al
    // tener @Target(ElementType.TYPE) podría haber sido aplicada, además de clases,
    // sobre enumerados, interfaces o incluso sobre tipos anotación
    if ((declaration instanceof ClassDeclaration)
        && !(declaration instanceof EnumDeclaration)) {

        // down-casteamos a declaración de clase
        ClassDeclaration classDecl = (ClassDeclaration) declaration;

        // actualizamos y mostramos las estadísticas sumando la clase
        sumarClaseYMostrarEstadisticas(classDecl);

        // para cada una de las declaraciones de métodos de la clase...
        for (MethodDeclaration methodDecl : classDecl.getMethods()) {

            // actualizamos y mostramos las estadísticas sumando el método
            sumarMetodoYMostrarEstadisticas(methodDecl);
        }
    }
}
```

La salida de este procesamiento de anotaciones es la información esperada, según se va actualizando con la contabilización de cada nueva clase o método (nótese que el orden en que se contabilizan las clases es indeterminado, tomándose antes la clase 2 antes que la clase 1):

```
Note: Clase encontrada: ClaseAnotadaAnotacionEjemploVisitor2 >>> Clases = 1 / Métodos: 0 / Media: 0.0
Note: Método encontrado: m1() >>> Clases = 1 / Métodos: 1 / Media: 1.0
Note: Método encontrado: m2() >>> Clases = 1 / Métodos: 2 / Media: 2.0
Note: Método encontrado: m3() >>> Clases = 1 / Métodos: 3 / Media: 3.0
Note: Método encontrado: m4() >>> Clases = 1 / Métodos: 4 / Media: 4.0
Note: Método encontrado: m5() >>> Clases = 1 / Métodos: 5 / Media: 5.0
Note: Clase encontrada: ClaseAnotadaAnotacionEjemploVisitor1 >>> Clases = 2 / Métodos: 5 / Media: 2.0
Note: Método encontrado: m1() >>> Clases = 2 / Métodos: 6 / Media: 3.0
Note: Método encontrado: m2() >>> Clases = 2 / Métodos: 7 / Media: 3.5
Note: Método encontrado: m3() >>> Clases = 2 / Métodos: 8 / Media: 4.0
```

Como en el caso de este ejemplo lo que queremos es procesar un tipo anotación sobre clases, podríamos implementar un Visitor redefiniendo el método **visitClassDeclaration**:

```
// iteramos sobre las declaraciones anotadas a ser procesadas
for (Declaration declaration : env.getDeclarationsAnnotatedWith(annotationTypeDecl)) {

    // PROCESAMIENTO DE LA DECLARACIÓN (CON VISITOR SimpleDeclarationVisitor)

    declaration.accept(new SimpleDeclarationVisitor() {

        @Override
        public void visitClassDeclaration(ClassDeclaration classDecl) {

            // CUIDADO: si es un tipo enumerado, no debemos tenerlo en cuenta
            if (classDecl instanceof EnumDeclaration) return;

            // actualizamos y mostramos las estadísticas sumando la clase
            sumarClaseYMostrarEstadisticas(classDecl);

            // para cada una de las declaraciones de métodos de la clase...
            for (MethodDeclaration methodDecl : classDecl.getMethods()) {

                // actualizamos y mostramos las estadísticas sumando el método
                sumarMetodoYMostrarEstadisticas(methodDecl);
            }
        }
    });
}

} // for Declaration
```

Ahora que el método que realiza el procesamiento reside en **visitClassDeclaration** dicho método sólo será invocado desde declaraciones de **ElementType.TYPE**, ya que el Visitor se invoca sobre declaraciones de ese tipo. Por ello, hay que tener cuidado si queremos excluir del cómputo a los enumerados (recordemos que son una especialización de las clases). Ya no es necesario comprobar que la declaración es **instanceof ClassDeclaration**, pero sí que sigue siendo necesario comprobar que no es un enumerado y, en dicho caso, retornar del método.

La salida de este procesamiento ofrece los mismos resultados totales que la versión anterior sin Visitor y exhibe el mismo comportamiento en todos los aspectos (incluyendo el orden indeterminista al captar las declaraciones de clase, que en esta ejecución concreta, sí ha salido ordenada, tomándose primero la clase anotada 1 y luego la clase anotada 2):

```
Note: Clase encontrada: ClaseAnotadaAnotacionEjemploVisitor1 >>> Clases = 1 / Métodos: 0 / Media: 0.0
Note: Método encontrado: m1() >>> Clases = 1 / Métodos: 1 / Media: 1.0
Note: Método encontrado: m2() >>> Clases = 1 / Métodos: 2 / Media: 2.0
Note: Método encontrado: m3() >>> Clases = 1 / Métodos: 3 / Media: 3.0
Note: Clase encontrada: ClaseAnotadaAnotacionEjemploVisitor2 >>> Clases = 2 / Métodos: 3 / Media: 1.0
Note: Método encontrado: m1() >>> Clases = 2 / Métodos: 4 / Media: 2.0
Note: Método encontrado: m2() >>> Clases = 2 / Métodos: 5 / Media: 2.5
Note: Método encontrado: m3() >>> Clases = 2 / Métodos: 6 / Media: 3.0
Note: Método encontrado: m4() >>> Clases = 2 / Métodos: 7 / Media: 3.5
Note: Método encontrado: m5() >>> Clases = 2 / Métodos: 8 / Media: 4.0
```

Aunque, a nivel de código parece que sólo nos hemos ahorrado el casting y poco más, los beneficios estructurales y de separación del código fuente serán mayores cuantos más tipos de declaraciones necesitemos procesar de forma separada, como veremos más adelante.

Diferentes tipos de Visitor: SimpleDeclarationVisitor y DeclarationScanner

En el ejemplo anterior hemos utilizado una instancia de `SimpleDeclarationVisitor` para visitar todas las declaraciones de clase que queríamos procesar. Esta clase es **la implementación más sencilla de la interfaz DeclarationVisitor y se limita a visitar única y exclusivamente las declaraciones del elemento dado por sus métodos visit_X**, teniendo en cuenta la jerarquía de herencia de las mismas. Por ejemplo, el visitar una clase se implementa así:

```
public void visitClassDeclaration(ClassDeclaration d) {  
    visitTypeDeclaration(d);  
}
```

`visitClassDeclaration` llama a `visitTypeDeclaration` porque una `ClassDeclaration` es, según el diseño dado por la Mirror API, una especialización de una `TypeDeclaration`. Esto se hace para mantener la coherencia conceptual impuesta por la jerarquía de herencia de los diferentes tipos de declaraciones existentes. Así, una llamada a `visitClassDeclaration`, viajando hacia arriba por el árbol de la jerarquía de herencia, provoca (en su implementación por defecto) la siguiente cadena de llamadas:

```
visitClassDeclaration → visitTypeDeclaration → visitMemberDeclaration → visitDeclaration
```

Por otra parte, tenemos la clase `DeclarationScanner` (“rastreador de declaraciones”). **DeclarationScanner es también una implementación de DeclarationVisitor que visita la declaración actual, así como también las subdeclaraciones contenidas en la misma.** Por ejemplo, el rastreo de una clase se implementa así:

```
public void visitClassDeclaration(ClassDeclaration d) {  
    d.accept(pre);  
  
    for(TypeParameterDeclaration tpDecl: d.getFormalTypeParameters()) {  
        tpDecl.accept(this);  
    }  
  
    for(FieldDeclaration fieldDecl: d.getFields()) {  
        fieldDecl.accept(this);  
    }  
  
    for(MethodDeclaration methodDecl: d.getMethods()) {  
        methodDecl.accept(this);  
    }  
  
    for(TypeDeclaration typeDecl: d.getNestedTypes()) {  
        typeDecl.accept(this);  
    }  
  
    for(ConstructorDeclaration ctorDecl: d.getConstructors()) {  
        ctorDecl.accept(this);  
    }  
  
    d.accept(post);  
}
```

Las variables de instancia `pre` y `post` de las clases `DeclarationScanner` son, a su vez, instancias de `DeclarationVisitor`, típicamente `SimpleDeclarationVisitor`, que dictan el procesamiento realizado antes (`pre`) y después (`post`) de visitar las subdeclaraciones.

En la implementación de `DeclarationScanner` puede verse como también se sube hacia arriba en la jerarquía de herencia, pero sólo en los tipos de declaración que están más abajo en la jerarquía y sólo hasta llegar a “tipos de declaración contenedores” que pueden contener otras subdeclaraciones. No se sube en la jerarquía de tipos de declaraciones en las visitas de paquetes, tipos (interfaces y clases), clases, ni ejecutables (constructores y métodos).

En base a lo anterior, una llamada a `visitClassDeclaration` de `DeclarationScanner` provoca (en su implementación por defecto), a su vez, la siguiente cadena de llamadas:

```
visitClassDeclaration
→ visitClassDeclaration del DeclarationVisitor dado por pre (Visitor de preprocesamiento)
→ visitTypeParameterDeclaration del DeclarationScanner para las subdeclaraciones de tipos parámetro
→ visitFieldDeclaration del DeclarationScanner para las subdeclaraciones de campos
→ visitMethodDeclaration del DeclarationScanner para las subdeclaraciones de métodos
→ visitTypeDeclaration del DeclarationScanner para las subdeclaraciones de tipos
→ visitConstructorDeclaration del DeclarationScanner para las subdeclaraciones de constructores
→ visitClassDeclaration del DeclarationVisitor dado por post (Visitor de postprocesamiento)
```

Hay que tener en cuenta que algunas de las llamadas realizadas podrían no sólo crear una cadena de llamadas para subir hacia arriba según la jerarquía de herencia, como pasaba en `SimpleDeclarationVisitor`, si no que podrían también conllevar algunos mínimos niveles de recursión. Por ejemplo, en el caso de hubiera una clase interna, `visitTypeDeclaration` invocará a `visitClassDeclaration` para dicha clase interna, creando una recursión que normalmente tendrá a lo sumo uno o dos niveles de profundidad, ya que normalmente los desarrolladores no suelen anidar clases a mayor profundidad.

Como podemos deducir, a efectos prácticos, **lo que la clase `DeclarationScanner` nos ofrece es la posibilidad de “transportar” instancias de `DeclarationVisitor` (pensadas para una declaración concreta) a través de toda la jerarquía de declaraciones**, dandonos la posibilidad adicional de elegir si queremos ejecutar el procesamiento dado por dichas instancias de `DeclarationVisitor` antes, con el Visitor de preprocesamiento (`pre`), y/o después, con el Visitor de postprocesamiento (`post`), de visitar las subdeclaraciones de la declaración que se está visitando en cada momento.

Finalmente, dado que el orden en que se visiten las subdeclaraciones puede tener mucha importancia, además de `DeclarationScanner`, que visita las subdeclaraciones sin ningún orden en particular, la Mirror API ofrece la clase `SourceOrderDeclarationScanner`, que visitará las subdeclaraciones en un orden lo más próximo posible al del código fuente. Instancias de ambas clases se pueden obtener a través de los métodos factoría estáticos de la clase `DeclarationVisitors` del paquete `util`.

En el siguiente ejemplo, la última variante del ejemplo del cómputo de estadísticas, utilizaremos precisamente `SourceOrderDeclarationScanner` para que los métodos se visiten siempre en el mismo orden. Nótese que, aunque `SourceOrderDeclarationScanner` garantiza que las subdeclaraciones de las clases se visitan en el orden más aproximado a cómo aparecen en el código fuente, las declaraciones de las propias clases en sí pueden ser captadas en un orden diferente cada vez. `SourceOrderDeclarationScanner` es determinista al nivel de las subdeclaraciones, pero el problema está en que `getDeclarationsAnnotatedWith` devuelve las declaraciones en un orden indeterminado.

Ejemplo: SourceOrderDeclarationScanner sobre decl. clases y métodos

En ambas implementaciones del ejemplo anterior del cómputo de estadísticas, sólo visitábamos declaraciones de clase, a pesar de que se realizaba procesamiento sobre los métodos de dichas declaraciones de clase mediante un bucle **for**. Esto era así porque, como hemos visto, la clase **SimpleDeclarationVisitor** sólo visita las declaraciones sobre las que se invoca su método **accept** y ninguna más. Sin embargo, sabiendo que **DeclarationScanner** sí visitaría tanto la declaración de la clase como, entre otras, todas las subdeclaraciones de sus métodos, podríamos sustituir el **for** concretamente por un **SourceOrderDeclarationScanner**.

Como el **DeclarationScanner** “transportará” el **SimpleDeclarationVisitor** a todas las subdeclaraciones de métodos de las clases visitadas, sólo hay que ubicar el código de procesamiento de los métodos en **visitMethodDeclaration** de nuestro **SimpleDeclarationVisitor** y utilizarlo, por ejemplo, como Visitor de preprocesamiento:

```
// iteramos sobre las declaraciones anotadas a ser procesadas
for (Declaration declaration : env.getDeclarationsAnnotatedWith(annotationTypeDecl)) {

    // PROCESAMIENTO DE LA DECLARACIÓN (CON VISITOR DeclarationScanner)

    declaration.accept(DeclarationVisitors.getSourceOrderDeclarationScanner(
        new SimpleDeclarationVisitor() { // Visitor de preprocesamiento

            @Override
            public void visitClassDeclaration(ClassDeclaration classDecl) {

                // CUIDADO: si es un tipo enumerado, no debemos tenerlo en cuenta
                if (classDecl instanceof EnumDeclaration) return;

                // actualizamos y mostramos las estadísticas sumando la clase
                sumarClaseYMostrarEstadisticas(classDecl);
            }

            @Override
            public void visitMethodDeclaration(MethodDeclaration methodDecl) {

                // actualizamos y mostramos las estadísticas sumando el método
                sumarMetodoYMostrarEstadisticas(methodDecl);
            }
        },
        DeclarationVisitors.NO_OP)); // Visitor de postprocesamiento
    } // for Declaration
```

El **DeclarationScanner** se ha encargado de llevar el **SimpleDeclarationVisitor** a las subdeclaraciones de métodos de las clases, y la salida es la misma que antes:

```
Note: Clase encontrada: ClaseAnotadaAnotacionEjemploVisitor2 >>> Clases = 1 / Métodos: 0 / Media: 0.0
Note: Método encontrado: m1() >>> Clases = 1 / Métodos: 1 / Media: 1.0
Note: Método encontrado: m2() >>> Clases = 1 / Métodos: 2 / Media: 2.0
Note: Método encontrado: m3() >>> Clases = 1 / Métodos: 3 / Media: 3.0
Note: Método encontrado: m4() >>> Clases = 1 / Métodos: 4 / Media: 4.0
Note: Método encontrado: m5() >>> Clases = 1 / Métodos: 5 / Media: 5.0
Note: Clase encontrada: ClaseAnotadaAnotacionEjemploVisitor1 >>> Clases = 2 / Métodos: 5 / Media: 2.0
Note: Método encontrado: m1() >>> Clases = 2 / Métodos: 6 / Media: 3.0
Note: Método encontrado: m2() >>> Clases = 2 / Métodos: 7 / Media: 3.5
Note: Método encontrado: m3() >>> Clases = 2 / Métodos: 8 / Media: 4.0
```

10.3.3.- apt (annotation processing tool).

10.3.3.1.- Descripción de apt.

apt (annotation processing tool = herramienta de procesamiento de anotaciones) es el comando ejecutable que, a través de sus opciones de línea de comandos, será el encargado de llevar a cabo todo el procesamiento de anotaciones.

Una vez lanzada una línea de comandos de apt completamente configurada, se llevarán a cabo las distintas fases del procesamiento de anotaciones. Inicialmente, **apt** determinará qué procesadores se encuentran disponibles a través de un proceso de descubrimiento (“*discovery*”) de las factorías disponibles, en base a las anotaciones presentes en los ficheros a procesar.

Una vez que **apt** ha emparejado las factorías de procesadores de anotación disponibles con los tipos anotación a procesar, realiza una ronda de procesamiento, que puede dar lugar a otras sucesivas si se va generando nuevo código fuente que, a su vez, sea necesario procesar.

Al llegar a la ronda final de procesamiento de anotaciones (la que sigue a la última que no haya generado nuevo código fuente a procesar), se lanzan los procesadores de anotaciones que correspondan por última vez y, una vez acabada la ronda final, todos los ficheros de código fuente, los iniciales y los generados, se pasan al compilador para su compilación definitiva.

A grandes rasgos, **su forma de funcionar hace que se pueda considerar a apt como una especie de sofisticado preprocesador**, ya que sigue una estrategia parecida a preprocesadores existentes para los compiladores de muchos otros lenguajes.

apt llevaría a cabo el papel habitual de un preprocesador, procesando el código inicial antes de ser enviado finalmente al compilador, una vez ha sido terminado de “preparar”. **Sin embargo, a diferencia de un preprocesador tradicional, el preprocesamiento que realiza apt no es fijo, si no que viene dado por la lógica implementada por cada procesador de anotaciones.**

De hecho, **el principal uso previsto para apt junto con los procesadores de anotaciones era muy similar al asignado tradicionalmente a los preprocesadores: la generación de ficheros derivados**. Esto se conseguía anotando toda una base de código fuente con ciertas anotaciones con el objetivo de crear la meta-information necesaria sobre dicho código fuente que permitiera la generación automática de ficheros derivados (nuevo código fuente, ficheros de clase, descriptores de despliegue, ficheros de configuración, etcétera). Estos ficheros derivados tendrían la ventaja fundamental de que, al ser regenerados con cada compilación, siempre estarán sincronizados y mantendrán la consistencia con el código original, cambiando también a la vez que lo hiciera el código original, lo que reduce dramáticamente el coste de mantenimiento de las aplicaciones.

No obstante, **apt** también puede utilizarse para llevar a cabo tareas más generales sobre el código fuente utilizando “procesadores universales”, que pueden procesar todo el código fuente, independientemente de que esté anotado o no. Posteriormente, con el paso de los años, también se le irá concediendo progresivamente una mayor importancia a la capacidad del procesamiento de anotaciones para realizar validación compleja sobre el código fuente, gracias a la capacidad de los procesadores para emitir errores de compilación.

10.3.3.2.- Funcionamiento de apt.

En este apartado vamos a entrar en algo más de detalle sobre el funcionamiento interno de **apt** a lo largo de sus diferentes fases.

Lanzamiento de apt mediante línea de comando

El procesamiento de anotaciones en J2SE 1.5 comienzan con una línea de comando **apt**. Se pueden indicar opciones como la factoría de procesadores de anotaciones a utilizar o, si son varias, las rutas donde buscar procesadores de anotaciones, los ficheros de código fuente que se desean procesar, el directorio de salida para los ficheros generados, etcétera.

Supongamos que lanzamos **apt** con la siguiente línea de comando:

```
apt -cp CLASSPATH .\src\paquete\ClaseAnotada.java
```

en la que se hace referencia a CLASSPATH, una variable de entorno que contiene el *classpath*, es decir, la ruta de clases necesarias para la compilación. Y después **la lista de ficheros sobre los que se realizará el procesamiento de anotaciones**, que en este caso es sólo una por simplicidad. Por supuesto, esta es una línea de comando muy sencilla, utilizada a modo ilustrativo, pero si se hace un uso más completo de todas las opciones ofrecidas por **apt**, el comando puede ser bastante más largo y complicado. Puede consultarse la referencia completa de opciones disponibles para **apt** en el subapartado “Opciones de línea de comando de **apt**”).

Búsqueda de tipos anotación a procesar

Lo primero que hace **apt** es realizar una **búsqueda de tipos anotación** presentes en el código fuente que se va a procesar. En este caso, se analizará el texto del código fuente del fichero **ClaseAnotada.java** para encontrar qué anotaciones contiene. Supongamos que **apt** encuentra varias anotaciones del tipo anotación **@AnotacionEjemplo1** y **@AnotacionEjemplo2**.



Descubrimiento de factorías de procesadores de anotaciones

A continuación, cada vez que tenga que encontrar un procesador de anotaciones para un determinado tipo anotación, **apt** llevará a cabo el proceso de descubrimiento (en inglés: *discovery*) de las factorías de procesadores de anotación que puedan crear instancias de procesadores que procesen dichos tipos anotación. El **descubrimiento de las factorías de procesadores de anotaciones** depende de cómo se configuren las opciones la línea de comandos de **apt**:



Procedimientos de Descubrimiento de factorías en función de las opciones de apt	
Opción	Procedimiento de Descubrimiento de factorías
Ninguna (por defecto)	Ruta de búsqueda del descubrimiento de factorías: CLASSPATH. Ficheros META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory
-factorypath FACTORYPATH	Ruta de búsqueda del descubrimiento de factorías: FACTORYPATH. Ficheros META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory
-factory CLASE_FACTORIA	No se realiza el procedimiento de descubrimiento de factorías. Se utiliza directamente una instancia de CLASE_FACTORIA.

El descubrimiento de factorías se hace vía [Service Provider Interface](#) (SPI) leyendo los ficheros `META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory` de las rutas de búsqueda (`CLASSPATH` o `FACTORYPATH`, según el caso), así como esos mismos ficheros dentro de los ficheros `.jar` especificados por esas rutas de búsqueda. Recordemos que una ruta de búsqueda puede ser una lista de directorios, pero también se pueden incluir ficheros `.jar`, que tienen su propia estructura de ficheros y dentro de los cuales también se buscará.

Los ficheros `META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory` que se vayan encontrando durante el proceso de descubrimiento, deberán tener una línea por cada factoría de procesadores que implemente la interfaz `AnnotationProcessorFactory`. La línea simplemente indicará el nombre canónico de la clase factoría para que `apt` pueda crear instancias de la misma, cuando necesite interrogarlas por los tipos anotación que procesan. Por ejemplo, el siguiente fichero hará que `apt` descubra las tres factorías cuyos nombres canónicos se especifican en él, uno por cada línea del fichero:

Fichero META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory
<code>anotaciones.ejemplos.procesamiento.java5.mirror.AnotacionElementoClaseProcessorFactory</code>
<code>anotaciones.ejemplos.procesamiento.java5.visitor.AnotacionEjemploVisitorProcessorFactory</code>
<code>anotaciones.ejemplos.procesamiento.java5.pruebas.AnotacionPruebasProcessorFactory</code>

Si se utiliza la opción `-factory nombre.paquete.EjemploAnnotationProcessorFactory`, no se realiza el proceso descubrimiento y se toma como única factoría disponible la indicada por dicha opción, que en el caso del ejemplo sería `EjemploAnnotationProcessorFactory`.

Emparejamiento factorías de proc. anotaciones ↔ tipos anotación

Una vez que `apt` ha determinado qué anotaciones tiene que procesar (en nuestro caso: `@AnotacionEjemplo1` y `@AnotacionEjemplo2`) y con qué factorías de procesadores cuenta (pongamos que en nuestro caso fuera sólo `EjemploAnnotationProcessorFactory`), realiza un emparejamiento entre factorías de procesadores y tipos anotación a procesar.

El procedimiento de emparejamiento es el siguiente:

- (1) `apt` solicita a cada factoría una lista de los tipos anotación que procesan.
- (2) `apt` pedirá a la factoría un procesador si la factoría procesa alguno de los tipos anotación.
- (3) El proceso continúa hasta que todos los tipos anotación tienen una factoría asignada o ya no quedan más factorías disponibles que asignar.

Es decir, que las factorías son consultadas para determinar los tipos anotación que procesan. Si una factoría procesa alguno de los tipos anotación presentes, dicho tipo anotación se considera como “reclamado” (en inglés: “claimed”) por la factoría que es capaz de procesarlo. Esto continúa hasta que todos los tipos anotación son reclamados o ya no quedan más factorías.

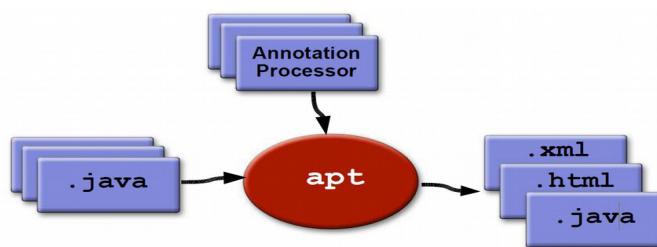
CUIDADO: Recordemos que los procesadores universales reclamarán para sí mismos el procesamiento de todos los tipos anotación (“*”). Esto provocará que, si se mezclan en la ruta de descubrimiento procesadores normales y procesadores universales, los procesadores normales no lleguen a ejecutarse, ya que `apt` dejará de buscar más factorías en cuanto encuentre la primera factoría que le informe de que sus procesadores soportan “*”. Debido a esto, la forma de proceder recomendada con los procesadores universales es tenerlos aislados en un JAR propio y realizar llamadas concretas de `apt` para cada uno de ellos usando la opción `-factory` para seleccionar cada uno de ellos expresamente en cada llamada específica.

Ronda 1 de procesamiento de anotaciones

IMPORTANTE: Los procesos de búsqueda de tipos anotación a procesar en el código fuente, descubrimiento de factorías de procesadores de anotaciones y su emparejamiento ocurren en la fase inicial de todas las rondas de procesamiento. Únicamente se han tratado como apartados introductorios independientes previos a este para poder explicarlos en detalle.

Una vez que se han analizado los ficheros de código fuente, encontrado las anotaciones a procesar y emparejado sus tipos anotación con las factorías de los procesadores de anotaciones que las procesarán, que son tareas previas a toda ronda de procesamiento, **apt** estará en disposición de ejecutar los procesadores de anotación sobre las anotaciones encontradas.

Para llevar a cabo el procesamiento de anotaciones como tal, **apt** creará una instancia de cada procesador de anotaciones a ejecutar a través del método **getProcessorFor** de sus correspondientes factorías y a continuación llamará al método **process()** de cada uno de los procesadores para que estos realicen su procesamiento de anotaciones correspondiente.



Cuando todos y cada uno de los procesadores de anotaciones descubiertos y emparejados han finalizado la ejecución de su método **process()**, termina la “ronda 1” o “primera ronda” del procesamiento de anotaciones. Además, en el procesamiento de anotaciones J2SE 1.5, al finalizar cada ronda se dispara el evento **roundComplete**, sobre las clases que se hayan registrado como *listener* del evento e implementen la interfaz **RoundCompleteListener**.

Como veremos más adelante, puede haber más rondas de procesamiento, incluso, si se dan determinados errores en el diseño de los procesadores, podría crearse un procesamiento de anotaciones recursivo con infinitas rondas de procesamiento que no terminaría jamás.

Todas las rondas de procesamiento de anotaciones son iguales, salvo algún pequeño detalle de la ronda final y algún otro pequeño detalle de esta primera ronda. Los detalles específicos de la ronda final se tratan en su correspondiente apartado, pero aquí hay que comentar un detalle muy importante que ocurre únicamente en esta primera ronda de procesamiento, y que es muy importante para facilitar el procesamiento de anotaciones en las rondas posteriores: la **instanciación y persistencia de las clases factoría**.

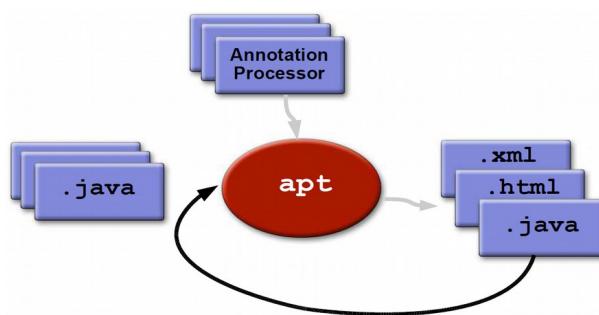
apt instancia las clases factoría en la primera ronda de procesamiento y las mantiene durante toda la duración del procesamiento, por lo que las instancias de las factorías son el lugar ideal para guardar cualquier información de estado o de contexto útil para el procesador, ya que se mantendrá intacta a lo largo de las sucesivas rondas de procesamiento. Esta información de estado puede guardarse fácilmente como variables de clase estáticas de la clase factoría o como variables de instancia. No obstante, en este último caso, será necesario establecer en el procesador una referencia a su objeto factoría en el cuerpo del método **getProcessorFor**, cuando la factoría construya el procesador.

Rondas adicionales de procesamiento de anotaciones (si procede)

El concepto de “ronda” (del inglés: “round”) surge debido al hecho de que los procesadores de anotaciones pueden generar nuevo código fuente. Y dicho nuevo código fuente generado ¡¡¡podría a su vez contener anotaciones que es necesario procesar!!!

Es decir, que mientras que haya procesadores de código que generen nuevo código fuente, el procesamiento de anotaciones no puede finalizar, debe continuar con una nueva “ronda de procesamiento” adicional, entrando en un proceso recursivo que no terminará hasta que una ronda de procesamiento no genere nuevo código fuente que procesar y se alcance por tanto la denominada “ronda final”.

Si se hace necesaria una ronda de procesamiento adicional debido a que algún procesador de anotaciones ha generado nuevo código fuente, `apt` pasará a realizar un proceso muy similar al de la primera ronda, pero esta vez lo que se tomará como argumento de entrada de código fuente a procesar será el nuevo código fuente recién generado en la ronda anterior, como se ilustra en el siguiente esquema:



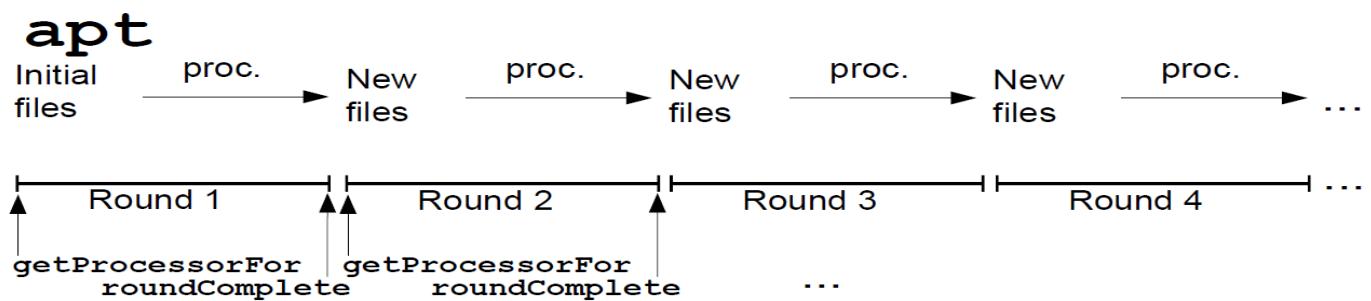
`apt` buscará las anotaciones existentes del nuevo código y emparejará los tipos anotación con las factorías de los procesadores adecuados de entre los disponibles, instanciándolos con el método `getProcessorFor` de su factoría y a continuación llamando al método `process()` de todos los procesadores para que procesen el nuevo código fuente generado en la ronda anterior.

NOTA: En las rondas adicionales de procesamiento, `apt` llamará al método `getProcessorFor` para todas las factorías que hayan proporcionado procesadores en alguna de las rondas anteriores, incluso si la factoría no crea procesadores para las anotaciones a procesar en esta ronda (es decir, incluso cuando sus procesadores no son necesarios). Esto da a la factoría la oportunidad de registrar un listener que podría utilizar para ser notificada de la finalización de las siguientes rondas. No obstante, este es un caso muy especial, y la mayoría de factorías simplemente devolverán el procesador identidad (`AnnotationProcessors.NO_OP`) en dicho caso.

Procesamiento de anotaciones: un proceso recursivo

El procesamiento de anotaciones puede, como hemos visto, convertirse en un **proceso recursivo** con un número desconocido de “rondas de procesamiento”, antes de llegar a terminar cuando se llega al final de una ronda sin que se haya generado nuevo código fuente (que sería pues el “caso base” de esta recursión).

Siendo así, el esquema temporal en forma de cronograma que sigue una ejecución de `apt` con múltiples rondas consecutivas de procesamiento (incluyendo todas sus sub-etapas: leer ficheros de entrada, buscar anotaciones, emparejar tipos anotación ↔ factorías, instanciar procesadores con `getProcessorFor` y procesar con `process`) tendría el siguiente aspecto:



Las rondas de procesamiento adicionales pueden ser ninguna, una o incluso, dado que es un proceso recursivo, podría generarse un “bucle infinito de procesamiento de anotaciones” si se diera el caso de que el diseño de los procesadores de anotaciones provocase que siempre se genere nuevo código fuente siempre en cada nueva ronda. Esto es así porque, en ese caso, no se estaría respetando el “caso base” de la recursión y podríamos convertirla en una recursión “no convergente” o “divergente”, es decir, que no se acerque al caso base recursivo y que, por tanto, no llegue a terminar jamás.

Así pues, podría darse un “bucle infinito de procesamiento de anotaciones”, por ejemplo, si diseñáramos un procesador de anotaciones que siempre generase código fuente nuevo (`GeneradorEternoCodigoFuenteProcessor`). O también en el caso de que dos o más procesadores de anotaciones que generasen tipos anotación relacionados y se creara un lazo entre ellos:

Entrada: ClaseA anotada con una anotación `@GeneraA`

Ronda 1: `GeneradorClaseAProcessor` → genera nueva ClaseA1 anotada con `@GeneraB`

Ronda 2: `GeneradorClaseBProcessor` → genera nueva ClaseB1 anotada con `@GeneraA`

Ronda 3: `GeneradorClaseAProcessor` → genera nueva ClaseA2 anotada con `@GeneraB`

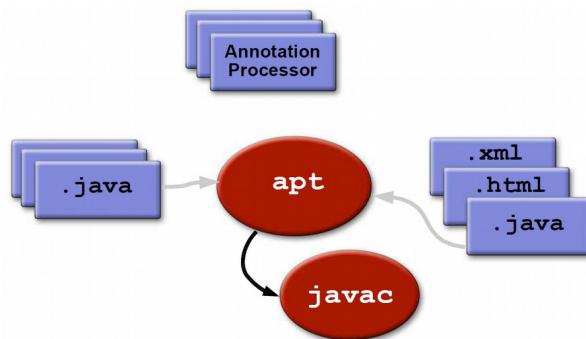
Ronda 4: `GeneradorClaseBProcessor` → genera nueva ClaseB2 anotada con `@GeneraA`

... *ad infinitum* ... (o más bien exactamente hasta que el entorno de ejecución arroje un `OutOfMemoryError`)

Ronda final del procesamiento de anotaciones

Una vez completada una ronda de procesamiento sin generar nuevo código fuente, se entra en la “ronda final”, donde **apt** lanza por última vez todos los procesadores para darles una oportunidad de que finalizan sus tareas (cierre de ficheros, liberación de recursos, etcétera). Una vez ejecutados todos los procesadores, se da por concluido el procesamiento como tal.

Terminado el procesamiento de anotaciones, **apt** realiza antes de terminar una llamada final a **javac** (el compilador de Java) para que compile todos los ficheros de código fuente, tanto los que había originalmente antes del procesamiento, como todos los nuevos ficheros de código que se hayan podido generar a lo largo de todas las rondas de procesamiento.



NOTA: La llamada final de **apt** al compilador Java puede no darse si se especifica la opción **-nocompile** en la línea de comandos. Esta opción puede ser de interés a la hora de depurar errores cometidos por los procesadores de anotaciones, ya que permite examinar la salida exacta de ficheros generada en las rondas de procesamiento de anotaciones, pudiendo ver así si el procesamiento de anotaciones está generando los ficheros adecuados y de la forma esperada.

Código de retorno de **apt**

El código de retorno (en inglés “return code” o “exit status”) de **apt** puede ser de mucha importancia en caso de que queramos programar scripts de procesos de compilación o de construcción de aplicaciones que usen **apt** para el procesamiento de anotaciones.

Si se invoca **javac** después de la última ronda de procesamiento, el código de retorno de **apt** será el que devuelva **javac**, ya que en **apt** delega totalmente en **javac** la compilación final de todos los ficheros generados durante el procesamiento de anotaciones.

Si no se invoca **javac** (debido a que se ha especificado la opción **-nocompile** de **apt**), **apt** devolverá un código de retorno **0** si no se produjeron errores durante su ejecución, ya sea por parte de la herramienta en sí o por parte de alguno de los procesadores de anotaciones.

NOTA: Operar sobre ficheros de código fuente incompletos o mal formados no es causa suficiente para que **apt** tenga un código de retorno distinto de **0**. Deberán de darse otro tipo de errores. En concreto, la entrada de ficheros de código fuente incompletos no provoca ningún error porque **apt** está diseñado para permitir que un procesador de anotaciones pueda corregir esos ficheros erróneos o incompletos. **apt** sólo emitirá un código de retorno erróneo si el error se detecta en la compilación final por parte de **javac**. Por este motivo, aunque a **apt** le entran ficheros mal formados, si los procesadores o la propia herramienta no arrojan ningún error, y se especifica la opción **-nocompile**, **apt** devolverá un código de retorno correcto **0**.

10.3.3.3.- Opciones de línea de comandos de apt.

Lo que sigue es una referencia completa de todas las opciones de línea de comando posibles para el comando `apt`. Es importante además tener en cuenta que `apt` **puede aceptar también todas las opciones de javac** (`javac -help` para más información), que serán pasadas al compilador tal cual después de la última ronda de procesamiento de anotaciones.

Opciones de línea de comandos de apt	
Opción	Descripción
<code>-classpath CLASSPATH</code>	Ruta de los ficheros clases de usuario de donde cargar las definiciones de las clases a emplear tanto en el procesamiento de anotaciones como al compilar. Puede incluir tanto directorios como ficheros <code>.jar</code> . De hecho, normalmente el <code>CLASSPATH</code> incluye la ruta base de los ficheros de clase compilados del usuario, así como tantas librerías <code>.jar</code> sean necesarias para sus dependencias. Si no se especifica <code>-factorypath</code> , es también la ruta por defecto para el descubrimiento de factorías de procesadores de anotaciones en los ficheros correspondientes: <code>META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory</code>
<code>-cp CLASSPATH</code>	
<code>-s DIR_FUENTES_GENERADAS</code>	Directorio raíz donde colocar los ficheros de código fuente generados durante el procesamiento de anotaciones. Cada fichero se guardará en el subdirectorio que corresponda a su respectivo paquete.
<code>-d DIR_DESTINO_CLASES</code>	Directorio raíz donde colocar los ficheros de clase generados al compilar todos los ficheros de código fuente existentes tras la última ronda de procesamiento (ficheros originales más los generados). Cada fichero de clase se guardará en el subdirectorio que corresponda a su respectivo paquete.
<code>-factorypath FACTORYPATH</code>	Ruta de los ficheros clases de usuario de donde realizar el descubrimiento de factorías de procesadores de anotaciones. Puede incluir tanto directorios como ficheros <code>.jar</code> . En la ruta, se buscan factorías en los ficheros correspondientes: <code>META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory</code>
<code>-factory CLASE_FACTORIA</code>	Especifica que no se realiza el procedimiento de descubrimiento de factorías, y que se utilizará como única factoría de procesadores la <code>CLASE_FACTORIA</code> dada.
<code>-source VERSION_JAVA</code>	Especifica que se quiere compatibilidad a nivel de código fuente con la versión de Java indicada. Por defecto, se tomará el nivel de código de la JDK a la que pertenezca la implementación de <code>apt</code> con la que estemos trabajando.
<code>-nocompile</code>	Hace que <code>apt</code> no llame al compilador para compilar los ficheros de código fuente generados tras la última ronda de procesamiento. Sirve para depuración.
<code>-Aclave[=valor]</code>	Opciones para los procesadores de anotaciones.
<code>-J<OPCIÓN></code>	Permite pasar opciones al entorno de ejecución Java donde va a lanzarse <code>apt</code> . De esta forma, podría ajustarse la memoria a utilizar durante la ejecución de <code>apt</code> y el resto de tipo de ajustes soportados. En cuanto al procesamiento de anotaciones, ya que las opciones para los procesadores de anotaciones que se pasan usando <code>-Aclave[=valor]</code> no se parsean correctamente, podemos elegir pasar dichas opciones como propiedades del entorno de ejecución usando la notación <code>-J-Dclave[=valor]</code> . Más información en el subapartado de este manual “Opciones para los procesadores desde línea de comandos”.
<code>-print</code>	Imprime la lista de tipos especificados. Si se especifica esta opción no se realiza ningún procesamiento de anotaciones ni compilación. Sirve para depuración.
<code>-version</code>	Imprime información de la versión de <code>apt</code> . Por ejemplo: <code>apt 1.7.0_05</code> .
<code>-help</code>	Imprime una sinopsis de las opciones estándar de <code>apt</code> .
<code>-x</code>	Imprime una sinopsis de las opciones no estándar de <code>apt</code> .
<code>-XListAnnotationTypes</code>	Lista los tipos anotación encontrados en el código. Sirve para depuración.
<code>-XListDeclarations</code>	Lista las declaraciones especificadas y las incluidas. Sirve para depuración.
<code>-XPrintAptRounds</code>	Imprime información de las rondas de procesamiento. Sirve para depuración.
<code>-XPrintFactoryInfo</code>	Imprime las anotaciones que procesa cada factoría. Sirve para depuración.
<code>-XclassesAsDecl</code>	Tratar ambos, ficheros de clases y fuente, como declaraciones a procesar. Normalmente sólo se tratan como declaraciones a procesar los ficheros fuente. Esta opción parece estar incluida sólo en JDK 7 (no está documentada antes).

Sintaxis general de `apt`

```
apt [-classpath CLASSPATH]
      [-s DIR_FUENTES_GENERADAS] [-d DIR_DESTINO_CLASES]
      [-factorypath FACTORYPATH] [-factory CLASE_FACTORY]
      [-source VERSION_JAVA] [-nocompile]
      [-Aclave[=valor] ...] [-J<OPCION> ...]
      [-print] [-version] [-help]
      [-X] [-XListAnnotationTypes] [-XListDeclarations]
      [-XPrintAptRounds] [-XPrintFactoryInfo] [-XclassesAsDecls]
      [opciones javac] FICHEROS_FUENTE [@FICHEROS_INLINE]
```

El orden es el mismo que el elegido en el cuadro de referencia de la página anterior, aunque en realidad es posible especificar las opciones en prácticamente cualquier orden, siempre que aparezcan antes de la lista de **FICHEROS_FUENTE**.

Todas las opciones están entre corchetes [] porque el único parámetro realmente obligatorio del comando `apt` es el de los **FICHEROS_FUENTE** a procesar. Todas las demás opciones no son obligatorias en absoluto porque, si son necesarias tienen valores por defecto si no se especifican, o son opciones que realizan acciones aisladas que no hay por qué especificar.

Al igual que con `javac`, se pueden especificar listas de ficheros fuente o de opciones a través de **@FICHEROS_INLINE**, esto es, ficheros de texto con una línea por fichero u opción.

Recomendaciones sobre el establecimiento de opciones para `apt`

Siempre que sea posible, se considera recomendable separar los diferentes directorios y rutas de búsqueda que entran en juego a lo largo de todo el proceso llevado a cabo por `apt`, de forma que las entradas al compilador estén claramente separadas de las salidas del compilador:

Rutas de directorios de ficheros para la entrada y salida de <code>apt</code>		
Opción	Dirección	Valor recomendado
<code>-cp CLASSPATH</code>	ENTRADA	Directorio raíz de las clases compiladas de la aplicación, así como de todos los ficheros .jar necesarios para la compilación.
<code>-s DIR_FUENTES_GENERADAS</code>	SALIDA	Suele ser un directorio llamado <code>generated_src</code> o similar.
<code>-d DIR_DESTINO_CLASES</code>	SALIDA	Suele ser un directorio llamado normalmente <code>classes</code> o <code>bin</code> .

Separar bien los directorio de entrada y salida de ficheros al compilador genera unas estructuras más higiénicas que cuando se mezclan varios tipos de ficheros, lo cual además puede traer problemas adicionales si los ficheros originales están bajo algún tipo de control de versiones de código fuente. Los ficheros derivados no deberían de estar bajo dicho control.

10.3.3.4.- Invocación de apt.

Hay muchas formas de invocar a **apt** para lanzar el proceso de procesamiento de anotaciones. En función de nuestras necesidades o de cómo se haya organizado el proceso de desarrollo y compilación de una aplicación concreta, puede venirnos mejor un método u otro.

Se ha tratado de ser exhaustivos y se explican todos los métodos posibles de invocación de **apt**. Aunque algunos de ellos carecerán de interés en un caso general, como las modalidades de invocaciones programáticas, pueden ser interesantes para el desarrollo de front-ends, wrappers o plugins que tengan que invocar el procesamiento de anotaciones.

De mayor utilidad práctica y de un uso mucho más extendido son la invocación a través de [Apache Maven](#), el sistema de administración de dependencias (“dependency management”) y construcción (“build”) de aplicaciones más usado en la actualidad en la comunidad Java.

Finalmente, se incluye una referencia al proceso de configuración de procesamiento de anotaciones en el popular IDE [Eclipse](#). Para saber del procedimiento de configuración necesario para el procesamiento de anotaciones en otros IDEs como [NetBeans](#), [IntelliJ IDEA](#) o cualquier otro de los [IDEs Java](#) disponibles, por favor, acuda a la documentación oficial de los mismos.

10.3.3.4.1.- Invocación de apt mediante línea de comandos.

Desde luego, la invocación directa de **apt** por línea de comandos, aunque es la forma más básica y estándar, es poco práctica a la hora de sobreponer un desarrollo de aplicaciones intenso. El medio más básico para invocarlo es desde el intérprete de comandos de nuestro sistema operativo: la ventana de comandos de Windows, una consola o terminal de GNU/Linux, etc.

Obviamente, para no tener que estar escribiendo continuamente el comando necesario, se definirán, como mínimo, unos ficheros de script que nos permitan lanzar **apt** de forma cómoda.

Ejemplo: Comando apt especificando múltiples opciones en Windows

Lo siguiente es un ejemplo de invocación de **apt** desde un fichero **.bat** de Windows:

```
apt -version -AopcionString="Valor String" -AopcionInteger=123 ^
-cp ".\target\classes;.\lib\apt-mirror-api.jar" ^
-factory anotaciones.ejemplos.procesamiento.java5.pruebas.EjemploProcessorFactory ^
.\src\main\java\anotaciones\ejemplos\procesamiento\java5\pruebas\PruebasMain.java
```

NOTA: El acento circunflejo ^ permite multilínea en ficheros **.bat** de Windows. Sólo hay que colocarlo al final de la línea y dejar al menos un espacio en blanco al inicio de la línea siguiente. Esto nos permite crear unos scripts de invocación escalonados en varias líneas mucho más sencillos de leer y, al facilitar que las líneas sean menos largas, sin tener que desplazarnos tanto con el scroll lateral a izquierda y derecha.

NOTA: El directorio donde se ubica **apt.exe**, normalmente **JDK_HOME\bin**, debe estar en la ruta de ficheros ejecutables del sistema operativo para que el comando funcione tal cual está escrito en este manual. En caso de que esto no sea así, puede incluirse la ruta de **apt.exe** en dicha ruta de ejecutables (en Windows: incluir **%JDK_HOME%\bin** como parte de la variable de entorno **Path**) o simplemente poner la ruta completa al directorio del comando, por ejemplo:

```
C:\Software\Desarrollo\Java\JSDK\JavaSE7\bin\apt.exe -version ... resto del comando ...
```

10.3.3.4.2.- Invocación de apt programáticamente como proceso.

Partiendo de que queremos realizar la invocación ejemplo de `apt` por línea de comandos, dicha llamada se podría recrear programáticamente usando las clases `ProcessBuilder` y `Process` del paquete `java.lang` usando un código similar al siguiente:

```
protected static void lanzarAPTProgramaticamenteComoProceso() throws IOException {  
  
    // ruta al directorio de la JDK donde está el ejecutable "apt.exe"  
    String aptRutaEjecutable =  
        "C:\\Software\\Desarrollo\\Java\\JSDK\\JavaSE7\\bin";  
  
    // para que las rutas relativas de la llamada original se resuelvan a las correctas  
    // (otra opción es pasarle siempre las rutas absolutas resueltas correspondientes)  
    String aptRutaBase =  
        "D:\\PFC\\Contenido\\3-Desarrollo\\EclipseWorkspace\\anotaciones-proc-java5";  
  
    String[] argumentos =  
    {  
        aptRutaEjecutable + "\\apt.exe",  
        "-version",  
        "-AopcionString=Valor String",  
        "-AopcionInteger=123",  
        "-cp",  
        ".\\target\\classes\\;..\\lib\\tools-jdk7.jar",  
        "-factory",  
        "anotaciones.ejemplos.procesamiento.java5.pruebas.AnotacionPruebasProcessorFactory",  
        ".\\src\\main\\java\\anotaciones\\ejemplos\\procesamiento\\java5\\pruebas\\PruebasMain.java"  
    };  
  
    // construimos el proceso en base a sus argumentos  
    ProcessBuilder processBuilder = new ProcessBuilder(Arguments.asList(argumentos));  
  
    // establecemos el directorio de trabajo como la ruta base donde está el proyecto  
    processBuilder.directory(new File(aptRutaBase));  
  
    // IMPORTANTE: redirigir la salida de error a la salida estándar para poder verla  
    processBuilder.redirectErrorStream(true);  
  
    // arrancamos el proceso  
    final Process proceso = processBuilder.start();  
  
    // imprimimos por consola la salida de la ejecución del proceso  
    BufferedReader br = new BufferedReader(new InputStreamReader(proceso.getInputStream()));  
    String lineaSalidaProceso;  
    while ((lineaSalidaProceso = br.readLine()) != null) {  
        System.out.println(lineaSalidaProceso);  
    }  
  
    // imprimimos el código de retorno  
    System.out.println("Código de retorno: " + proceso.exitValue());  
}
```

Consideraciones sobre el uso de `java.lang.Process` y `java.lang.ProcessBuilder`

- Las rutas relativas al directorio actual “.” no funcionan sin `aptRutaBase` como directorio.
- Las opciones que dejan espacios en blanco intermedios con su valor (como `-cp` o `-factory`), han de partirse en 2 para que el parser de `apt` las coja. Si no da un error de `invalid flag`.
- Al usar String's ya no habrá que utilizar comillas “” para marcar el inicio y el final de los valores que, por ejemplo, contengan espacios, pero sí **será necesario escapar la \ con **.

10.3.3.4.3.- Invocación de apt programáticamente (com.sun.tools.apt).

La invocación programática de **apt**, no como proceso, si no a través de la interfaz programática que modela la herramienta, es algo complicada de utilizar y de poco interés en general, pero la incluimos por completitud. El principal problema es que la API que permite la invocación no está considerada una API pública y carece totalmente de documentación.

Partiendo de que queremos realizar la invocación ejemplo de **apt** por línea de comandos, dicha llamada se podría invocar programáticamente con el siguiente fragmento de código:

```
protected static void lanzarAPTprogramaticamente() {  
  
    // necesitamos establecer la ruta base como propiedad "user.dir" de la Máq. Virtual  
    // para que las rutas relativas de la llamada original se resuelvan a las correctas  
    // (otra opción es pasarle siempre las rutas absolutas resueltas correspondientes)  
    String aptRutaBase =  
        "D:\\PFC\\Contenido\\3-Desarrollo\\EclipseWorkspace\\anotaciones-proc-java5";  
  
    String[] argumentos =  
    {  
        "-J", "-Duser.dir=" + aptRutaBase, // para que las rutas relativas a "." funcionen  
        "-version",  
        "-AopcionString=Valor String",  
        "-AopcionInteger=123",  
        "-cp",  
        ".\\target\\classes\\;.\\lib\\tools-jdk7.jar",  
        "-factory",  
        "anotaciones.ejemplos.procesamiento.java5.pruebas.AnotacionPruebasProcessorFactory",  
        ".\\src\\main\\java\\anotaciones\\ejemplos\\procesamiento\\java5\\pruebas\\PruebasMain.java"  
    };  
  
    // llamada programática a apt  
    int codigoRetorno = com.sun.tools.apt.Main.process(argumentos);  
  
    // imprimimos el código de retorno  
    System.out.println("Codigo de retorno: " + codigoRetorno);  
}
```

Consideraciones sobre el uso de com.sun.tools.apt.Main

- La clase que implementa la interfaz programática con **apt** es **com.sun.tools.apt.Main**. Esta clase está en **tools.jar** (en **JDK_HOME\lib**) de la JDK hasta la versión 7, por lo que sería necesario tener **tools.jar** en el **CLASSPATH**, algo poco recomendable y que puede traer muchos problemas si se mezclan ficheros **tools.jar** de distintas versiones de la JDK.
- Las rutas relativas al directorio actual “.” no funcionan directamente, hay que (1) establecer todas las rutas como absolutas o (2) establecer la propiedad “**user.dir**” de la Máquina Virtual Java al directorio que queremos que se use como directorio actual. En el ejemplo se opta por la opción (2) con **"-J", "-Duser.dir=" + aptRutaBase**. Nótese como la opción, que originalmente por línea de comandos sería **-J-Duser.dir=valor** se ha partido en 2 argumentos.
- Las opciones que dejan espacios en blanco intermedios con su valor (como **-cp** o **-factory**), así como la opción especial **-J**, como hemos visto, han de partirse en 2 argumentos.
- Al usar String's ya no habrá que utilizar comillas “” para marcar el inicio y el final de los valores que, por ejemplo, contengan espacios, pero sí **será necesario escapar la \ con **.

10.3.3.4.4.- Invocación de apt mediante Maven.

[Apache Maven](#) es hoy en día el sistema de administración de dependencias (“dependency management”) y construcción (“build”) de aplicaciones más utilizado en la comunidad Java. Aunque Maven es complejo de configurar y poner en marcha, una vez está todo configurado, automatiza de forma espectacular el proceso de construcción y exportación a contenedores JAR, WAR, EAR, etcétera. Su estandarización del proceso de construcción y la enorme productividad que ello ofrece han hecho de Maven un estándar de facto entre la comunidad de desarrollo Java. Para más detalles sobre Maven, puede consultarse su página web oficial apache.maven.org.

El hecho de que sea un sistema de construcción incluye la necesidad de relacionarse con el proceso de compilación llamando a `javac` y, por supuesto, con `apt`, nuestro “preprocesador sofisticado” para el procesamiento de anotaciones.

Maven puede invocar a `apt` utilizando uno de los plugins que extienden su funcionalidad, concretamente el `apt-maven-plugin` [<http://mojo.codehaus.org/apt-maven-plugin>]. Para ello, sólo hay que añadir al fichero `POM.xml` del proyecto Java en cuestión, dentro del elemento XML `<build> <plugins>`, el siguiente elemento `<plugin>`:

```
<!-- ejecuta directamente apt; goal "apt:process" -->
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>apt-maven-plugin</artifactId>
    <version>1.0-alpha-5</version>
    <configuration>
        <!-- el valor del elemento factory se mapea directamente al argumento -factory de apt,
            por lo que SÓLO SOPORTA 1 FACTORÍA; por tanto, lo dejamos comentado para que realice
            el descubrimiento a través de META-INF/services, que soporta múltiples factorías -->
        <!-- <factory>anotaciones.ejemplos.procesamiento.java5.EjProcessorFactory</factory>
    -->
    </configuration>
</plugin>
```

Cuando esté correctamente configurado, ejecutando Maven con el goal “`apt:process`” del plugin, podremos ver cómo se ejecutan todos los procesadores de anotaciones sobre el código del proyecto, mostrándose la salida del procesamiento de anotaciones por la salida de consola de Maven y, si todo se ha procesado correctamente y sin errores, veremos como el build de Maven termina correctamente con el clásico bloque `BUILD SUCCESS`:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.311s
[INFO] Finished at: Mon May 19 12:16:58 CEST 2014
[INFO] Final Memory: 9M/109M
[INFO] -----
```

CUIDADO: El goal “`apt:process`” de `apt-maven-plugin` ejecuta `apt` sólo para el procesamiento de anotaciones, pero no invoca `javac`. Es decir, que es equivalente a invocar a `apt` con la opción `-nocompile`. Esto no es problema, ya que a continuación se puede llamar al plugin de Maven que llama al compilador `javac`: el `maven-compiler-plugin`.

`apt-maven-plugin` soporta las opciones más importantes de línea de comandos de `apt` a través de distintos subelementos XML ubicados dentro de su elemento `<configuration>`. Para más información, puede consultarse la documentación del plugin en su página web oficial.

10.3.3.4.5.- Invocación de apt mediante Eclipse.

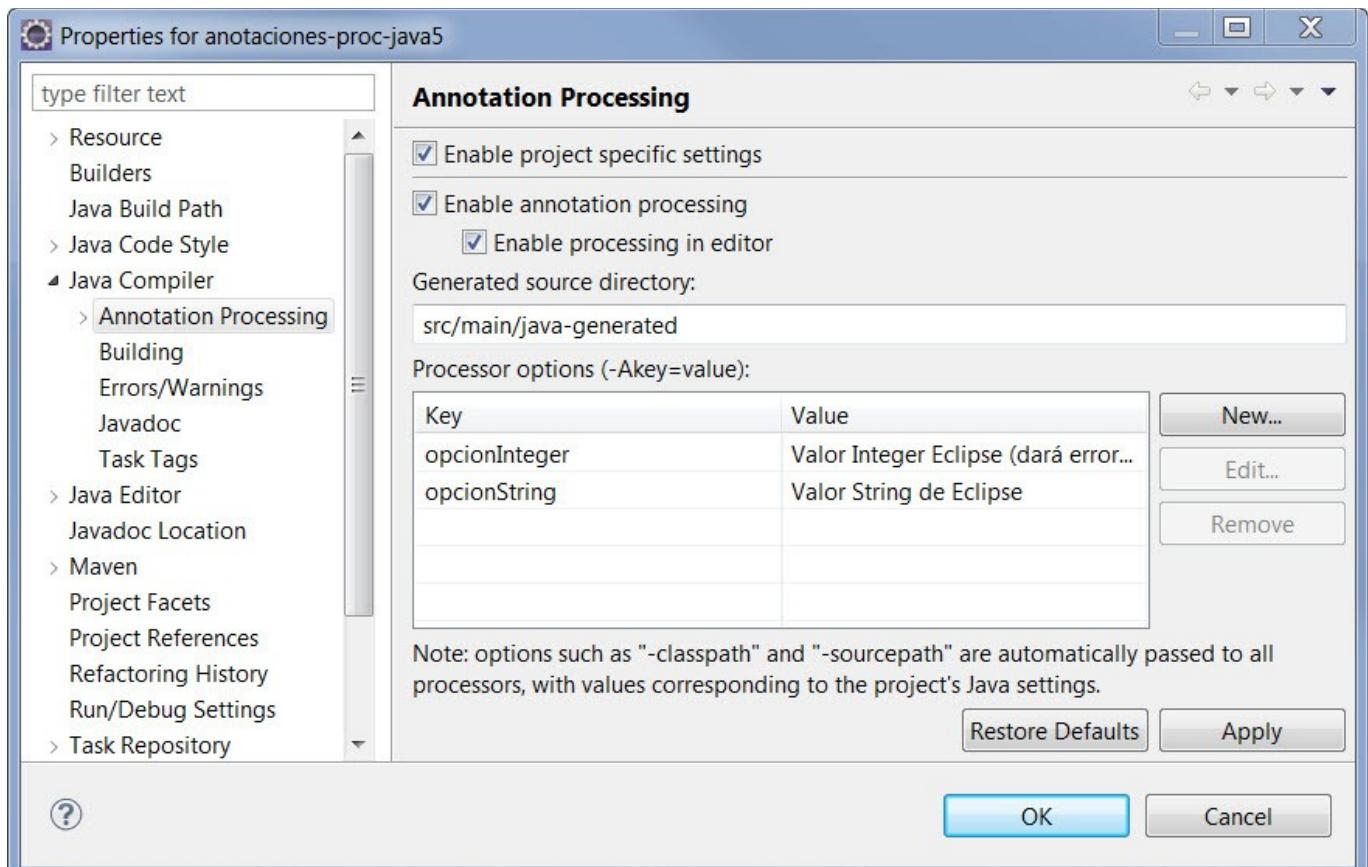
[Eclipse](#) se ha convertido en uno de los IDEs Java más extendidos en la actualidad entre la comunidad de desarrollo de la plataforma Java. Esto es debido a sus altas prestaciones como entorno de desarrollo a la hora de facilitar las tareas más tediosas y a su arquitectura extensible que, al igual que Maven, le permite añadir funcionalidades adicionales.

A la hora de invocar **apt**, precisamente Eclipse puede ser el IDE Java que comporte precisamente más problemas debido a que Eclipse implementa su propio compilador Java, distinto de la implementación de referencia: el clásico **javac** incluido en la JDK de Sun/Oracle. Esto se debe a que los diseñadores de Eclipse no estaban conformes con las limitaciones de **javac**, que no tenía soporte para algunas características que querían incorporar en Eclipse.

Funfamental fue que **javac** no soportara [compilación incremental](#), i.e., poder compilar sólo partes del código de una unidad de compilación (clase, módulo, paquete, etcétera), en lugar de la unidad completa. El compilador Java de Eclipse sí soporta compilación incremental con una operación llamada “reconciliación” o “reconciliar” (“reconcile”), lo cual da a Eclipse una mayor eficiencia, así como otra serie de características de las que otros IDEs Java carecen.

Así pues, el soporte de procesamiento de anotaciones en Eclipse puede tener algún bug circunstancial que, más tarde o más temprano, deberá ser corregido, debido a que todas las características que se introduzcan en la plataforma Java deben ser implementadas por Eclipse.

El soporte de procesamiento de anotaciones J2SE 1.5 en Eclipse se configura accediendo a las propiedades del proyecto, apartado “**Java Compiler**” / “**Annotation Processing**”:

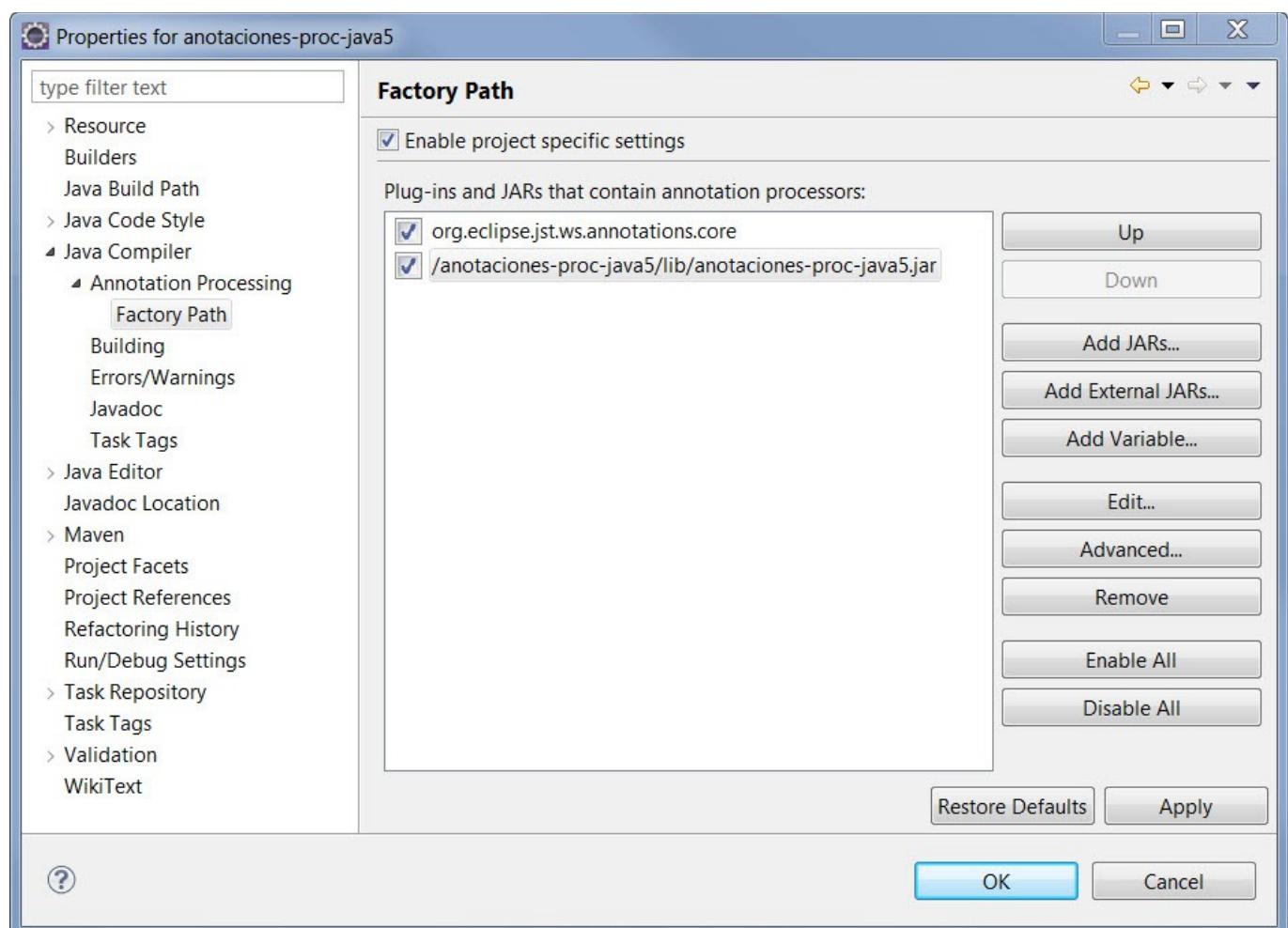


Marcando las opciones “**Enable project specific settings**” y “**Enable annotation processing**” activaremos el procesamiento de anotaciones para nuestro proyecto, pero no basta sólo con eso, claro. Hay algunas cosas más todavía que será necesario configurar.

Desde este mismo apartado, en “**Generated source directory**”, podemos ajustar el directorio donde queremos que se generen los nuevos ficheros de código fuente (equivalente a la opción **-s DIR_FUENTES_GENERADAS** de **apt**). El valor por defecto es “**.apt_generated**”, no el que hemos configurado nosotros (**CUIDADO**: al empezar por “.” podría ser que el directorio se oculte en algunas vistas del entorno). También disponemos de la posibilidad de pasar una lista de opciones a los procesadores de anotaciones con la notación **-Aclave[=valor]** introduciendo la clave y el valor directamente. Nótese que el valor establecido para la opción “**opcionInteger**” será un valor erróneo para el procesador, ya que no es un valor numérico.

Otra cosa importante que hay que tener en cuenta es que en la integración de **apt** en Eclipse, como se indica en la nota de la parte de abajo del cuadro de diálogo, las opciones de **-classpath** (y el menos utilizado **-sourcepath**, no confundir con **-s**) no podemos establecerlas nosotros, si no que se pasan automáticamente en función de la configuración del proyecto. Esto no debería ser un problema, ya que Eclipse permite configurar el **CLASSPATH** de un proyecto con bastante detalle en el apartado “Java Build Path” de sus propiedades.

El proceso de descubrimiento de factorías de procesadores de anotaciones es quizás lo que peor está resuelto en Eclipse debido a que sólo se busca en los archivos **.jar** configurados en el apartado “**Java Compiler**” / “**Annotation Processing**” / “**Factory Path**”:



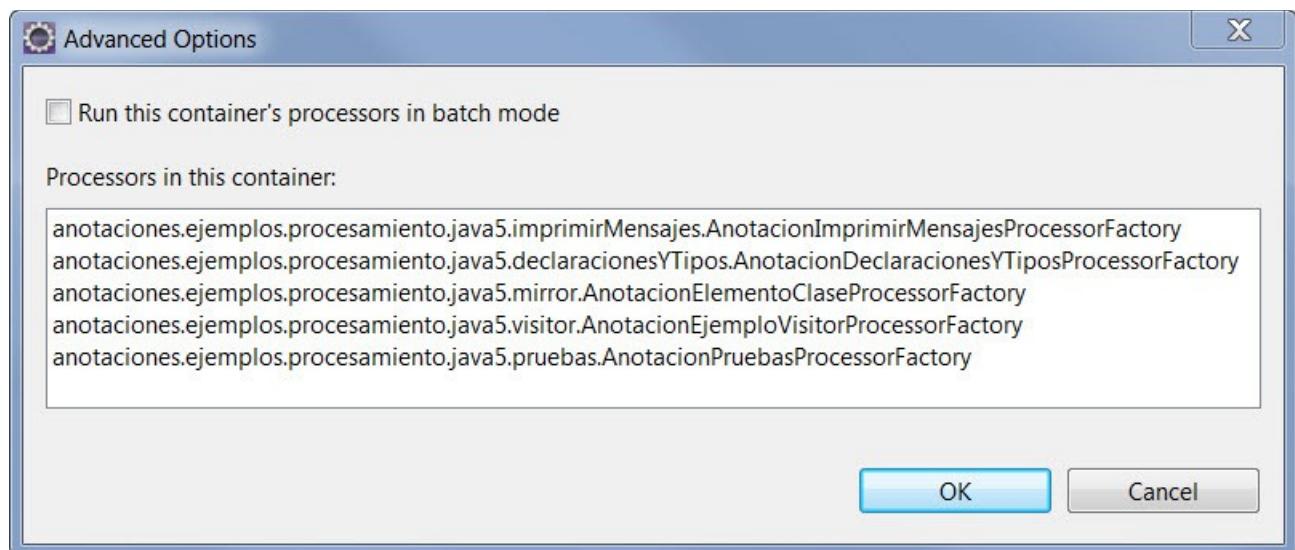
Esta falta de flexibilidad no es tal cuando invocamos a **apt** por línea de comandos o vía Maven, ya que en todos los casos excepto en Eclipse se lanza el proceso de descubrimiento de factorías implementado por **apt** que por defecto busca las factorías en el propio **CLASSPATH** indicado (en el caso de **apt**) o en el del proyecto (en el caso de Maven). No obstante, como hemos comentado, el compilador de Eclipse implementa su propio proceso de descubrimiento diferente al de **apt**, y en el que únicamente busca factorías en su propio **FACTORYPATH**, que serían los **.jar** que se añaden en este apartado “Factory Path”, y no busca en el **CLASSPATH** como tal del proyecto.

Que el proceso de descubrimiento de factorías de Eclipse no busque en el **CLASSPATH** del proyecto, sino sólo en el **FACTORYPATH** indicado, puede en principio parecer algo insignificante, pero tiene un inconveniente muy importante a nivel práctico de cara a trabajar en el desarrollo de procesadores de anotaciones: **es necesario exportar el propio proyecto a un .jar para que Eclipse descubra las factorías de nuestros propios procesadores. Pero lo peor de todo no es eso, si no que, si se modifica cualquier procesador, hay que volver a exportar el .jar ante cualquier modificación que se haga, por pequeña que sea.** Por este motivo, en el “Factory Path” de la imagen anterior aparece incluido el fichero **.jar** exportado del propio proyecto en cuestión (**anotaciones-proc-java5.jar**).

El tener que estar exportando el proyecto a **.jar** continuamente cada vez que se quiere probar cualquier mínima modificación sobre un procesador de anotaciones, como es lógico, es todo un engorro y, por ello, por ejemplo, **en el día a día del desarrollo de los ejemplos de procesamiento de anotaciones vinculados a este manual se ha venido trabajando con Maven**, que a este respecto resulta mucho más práctico al estar integrado también en Eclipse, pero, al invocar a **apt**, usa el mecanismo de descubrimiento de factorías de **apt** que sí busca en el **CLASSPATH** de Maven, que es el **CLASSPATH** del proyecto de Eclipse.

El problema anterior sólo es realmente grave si somos desarrolladores de procesadores de anotaciones, pero no si somos simplemente usuarios de dichos procesadores. No será problema si las factorías y sus respectivos procesadores de anotaciones que vamos a usar son de terceros, ya están desarrollados y probados, y nosotros únicamente vamos a usarlos.

Cada vez que incluyamos un **.jar** en el “Factory Path” de Eclipse es recomendable seleccionarlo y hacer click en el botón “**Advanced...**”. Esto nos mostrará un cuadro de diálogo “**Advanced Options**” con las factorías de procesadores encontradas en dicho fichero **.jar**.



Es una práctica recomendada hacer click en “Advanced...” y comprobar que, cuando añadimos un fichero .jar al “Factory Path”, Eclipse haya sido capaz de descubrir en dicho fichero todas las factorías de procesadores esperadas. Si no apareciera ninguna factoría, probablemente se deba a que el fichero .jar no está correctamente generado, o a que carezca del directorio de recursos /META-INF/services con el fichero que lista las factorías de procesadores J2SE 1.5, que recordemos que debe ser el siguiente:

/META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory

CUIDADO: No todos los .jar añadidos a la “Factory Path” han de contener factorías de procesadores de anotaciones necesariamente. La **FACTORYPATH** de Eclipse también tendrá que incluir los .jar de cualquiera otras librerías que los procesadores necesiten para reconocer las clases que procesan, así como para poder procesarlas. Por ejemplo: si añadiéramos un procesador de anotaciones que trabaje sobre las anotaciones de JPA ([Java Persistence API](#)), una API de persistencia que hace un uso muy intensivo de anotaciones, deberemos añadir al **FACTORYPATH** también el .jar con las clases de JPA propiamente dicha, o el procesamiento arrojará excepciones **ClassNotFoundException** para cada una de las clases de JPA, ya que no serán reconocidas. **En el ámbito de la ejecución del procesamiento de anotaciones, tal y como está implementado en Eclipse, es como si el CLASSPATH no existiera, sólo existe el FACTORYPATH.**

En “Advanced Options” también es posible marcar la opción “**Run this container's processors in batch mode**” (“Ejecutar los procesadores de este contenedor en modo por lotes”). Esta opción tiene que ver con lo que ya hemos comentado de que Eclipse trata de implementar la compilación incremental. Y, asimismo, ocurre que **la implementación del procesamiento de anotaciones en Eclipse también es incremental**.

Debido a que en la implementación de Eclipse el procesamiento de anotaciones es incremental, dicho procesamiento de anotaciones no transcurre en Eclipse exactamente igual que en **apt**. La opción “**Run this container's processors in batch mode**” hace que Eclipse ejecute los procesadores de anotaciones de una forma más parecida a la de la implementación de **apt**. Esta opción es, por tanto, una **opción de compatibilidad** que sirve para hacer funcionar que están tan vinculados y acoplados al comportamiento de **apt** que fallan si el procesamiento de anotaciones no se implementa tal y como lo hace **apt**, si no, por ejemplo, con los procesadores ejecutándose en paralelo o realizando su procesamiento de forma incremental.

La opción “**Run this container's processors in batch mode**” está pues pensada principalmente para procesadores que no pueden funcionar correctamente con procesamiento incremental: por ejemplo, porque descansen en variables estáticas que sólo se inicializan correctamente en la primera ronda del procesamiento, o que procesan todos sus ficheros en una ronda en lugar de un fichero por ronda o, en general, trabajan basándose en cualquier modo particular de comportamiento de **apt** que no se da en la implementación de Eclipse. Esto pasa, por ejemplo, precisamente con los procesadores de anotaciones de JPA dados por Sun/Oracle y es por ello que necesitarán que se active el modo por lotes para funcionar correctamente.

Cuando “Run this container's processors in batch mode” es activada, todo el procesamiento funcionará de manera que mejore la compatibilidad con apt: los procesadores se cargan al principio de la compilación, ejecutados sobre todos los tipos Java, y luego descargados, tal y como se hace en **apt**. Los procesadores que funcionan en modo por lotes sólo se ejecutarán en una compilación completa (que se puede forzar con la opción “Project” / “Clean...”) y no en una compilación incremental. Además, hay que comentar que **esta opción sólo afecta a los procesadores de anotaciones J2SE 1.5 que usan apt, y no afecta a los procesadores de anotaciones JSR 269.**

Finalmente, cuando hayamos completado toda la configuración de Eclipse correctamente, podremos ver que, si los procesadores se ejecutan y empiezan a emitir a warnings o errores, estos se mostrarán en la vista “**Problems**” o en la vista “**Error Log**”.

En la siguiente imagen puede apreciarse como, tras la configuración del procesamiento de anotaciones en el entorno y la ejecución de una compilación completa, aparecen ya algunas entradas en la vista “**Problems**” con el tipo “**Annotation Problem**” (diferentes de los errores del compilador, que son del tipo “Java Problem”). Nótese como se muestra el mensaje de error provocado al arrojar el procesador implementado una **NumberFormatException** debido a que se ha pasado un valor no numérico incorrecto para su opción “**opcionInteger**”.

Description	Path	Location	Type
⚠ opcionesLineaComandos = {opcionString=Valor String de Eclipse, -d=D:\PFC\Contenido\3-D /anotaciones-proc-java5/...}	/anotaciones-proc-java5/...	line 7	Annotation Problem
⚠ opciones del procesador: opcionString (leida) = Valor String de Eclipse	/anotaciones-proc-java5/...	line 7	Annotation Problem
⚠ opciones del procesador: opcionString (final) = Valor String de Eclipse	/anotaciones-proc-java5/...	line 7	Annotation Problem
⚠ opciones del procesador: opcionInteger (leida) = Valor Integer Eclipse (dará error al no ser un número)	/anotaciones-proc-java5/...	line 7	Annotation Problem
⚠ opciones del procesador: opcionInteger (final) = 1	/anotaciones-proc-java5/...	line 7	Annotation Problem
✖ opciones del procesador: opcionInteger: valor incorrecto: java.lang.NumberFormatException:	/anotaciones-proc-java5/...	line 7	Annotation Problem

IMPORTANTE: Los mensajes informativos emitidos con el método **printNotice** de la interfaz **Messager** de los procesadores de anotaciones pueden no mostrarse en ninguna de las dos vistas comentadas (aunque en algunas versiones de Eclipse podrían aparecer en la vista “**Error Log**”). Todo dependerá de la versión concreta de Eclipse de que dispongamos, ya que esto es un problema de la implementación concreta de Eclipse que está pendiente de ser corregido. Esperamos que en el futuro puedan aparecer los mensajes informativos en la vista “**Problems**” si así lo tenemos configurado para una mayor comodidad.

A diferencia de los mensajes informativos emitidos con **printNotice**, los warnings emitidos con **printWarning** y los errores emitidos con **printError** sí aparecen en la vista “**Problems**” del entorno. Por ello, sólo para poder mostrar los mensajes informativos de los procesadores por dicha vista (que es la más cómoda), hemos modificado temporalmente las llamadas originales de **printNotice** en el procesador sustituyéndolas por **printWarning**, pero no deberíamos ser nosotros quien tuviera que adaptar nuestro código, si no que deberán ser los desarrolladores de Eclipse lo que en algún momento corrijan este defecto.

10.3.3.5.- Fallos conocidos y limitaciones de **apt**.

A lo largo de este manual hemos mencionado ya algunas limitaciones y fallos conocidos del procesamiento de anotaciones, pero vamos a centralizar dicha información aquí para que, en caso de que se vaya a trabajar o ya se esté trabajando con **apt**, puedan conocerse de antemano los problemas de la herramienta. Algunos no están adecuadamente documentados y, debido a ello, han ido surgiendo con su uso y, verdaderamente, no son pocos ni insignificantes, por lo que este apartado ha tenido que incluirse cuando no estaba en absoluto previsto.

Hay que mencionar que, el hecho de que **apt** haya sido deprecado en favor de la API de procesamiento JSR 269, ha sido un factor determinante para que muchos de los errores se hayan dejado sin corregir, ya que el soporte de **apt** prácticamente fue abandonado con la salida de Java SE 6. La mayoría de estos bugs fueron corregidos para JSR 269, pero no para **apt**:

- **apt** no captura directamente las declaraciones sobre paquetes [[+info](#)]. Esto se debe a un bug heredado de **javac**, que no lee los ficheros package-info.java. Esto provoca que **apt** no sea capaz de ver las declaraciones de paquetes directamente (aunque sí indirectamente). Si se lanza **apt** sobre un package-info.java con la opción **-XListDeclarations** puede comprobarse como las listas de declaraciones están vacías (cuando deberían incluir la declaración del paquete). Aún así, las declaraciones de paquete pueden recuperarse llamando a **getPackage()** de cualquier instancia de **TypeDeclaration**. En ese caso, la declaración de paquete contiene las propiedades esperadas y responde correctamente a los métodos **getAnnotation** y **getAnnotationMirrors**. No obstante, si sólo se procesa una declaración de paquete sin otra declaración perteneciente a dicho paquete que nos permita “engancharla”, la declaración de dicho “paquete aislado” sería casi totalmente inaccesible, salvo quizás por el método **java.lang.Package.getPackages()**, pero este último no es totalmente confiable y, si se usa, sólo debería ser como “último recurso” para buscar precisamente “paquetes aislados” que no estén asociados a ninguna declaración.

- **apt** no puede procesar anotaciones sobre variables locales [[+info](#)]. Además, la Mirror API ni siquiera modela las declaraciones de variables (no existe la interfaz **VariableDeclaration**). Como ya se ha explicado, esta limitación se debe a que falta información en los ficheros de clase para poder permitirlo. Este tema se solucionó en Java SE 8 con la especificación JSR 308.

- **apt** no parsea correctamente las opciones tipo **-Aclave[=valor]** de los procesadores [[+info](#)]. Como se ha visto en el subapartado “Opciones para los procesadores desde línea de comandos”, **apt** parsea **-Aclave=valor** asignando como clave “**-Aclave=valor**” y como valor **null**. No obstante, también en dicho subapartado hemos visto que hay formas de lidiar con el problema.

- **apt** no siempre termina con un código de retorno erróneo ante errores de compilación [[+info](#)]. Como explicamos en “Funcionamiento de **apt**”, esto se debe a que **apt** permite la entrada de ficheros erróneos para que un procesador pudiera corregirlos. Sólo fallará si los ficheros siguen llegando erróneos a la compilación con **javac**. Esto hace que, si **apt** no arroja ningún error de procesamiento y los ficheros erróneos no se compilan al usar **-nocompile**, **apt** pueda terminar con un código de retorno correcto **0** en presencia de ficheros con errores de compilación.

- **apt** ignora la herencia de anotaciones en clases fuera del conjunto de declaraciones incluidas [[+info](#)]. Es decir, que si tenemos un tipo anotación **@AnotacionHeredable** que es **@Inherited** y que anota una clase **A**. Si **B** hereda de **A**, debe considerarse anotada con **@AnotacionHeredable**, aunque la anotación no esté directamente presente en la declaración de la clase **B**. Sin embargo, esto no ocurre así si al invocar a **apt** no se incluye el código fuente de ambas clases, **A** y **B**.

10.4.- Procesamiento J2SE 1.5: Paso a paso.

Hasta ahora, hemos visto una visión global del procesamiento de anotaciones J2SE 1.5, así como una inmersión detallada en cada uno de sus componentes para conocerlos uno a uno a fondo y tener una referencia completa de los mismos antes siguiera de empezar a desarrollar nuestros propios procesadores de anotaciones.

Pues bien, ya estamos en condiciones de ponernos manos a la obra y, por tanto, no vamos a retrasarlo más. En este apartado describiremos paso a paso, desde el principio hasta el final, el procedimiento habitual que se sigue para implementar el procesamiento de anotaciones.

10.4.1.- Análisis y Diseño del tipo anotación.

El requisito de implementar un procesamiento de anotaciones siempre surje de alguna necesidad concreta, que es analizada y cuyo análisis llega a la conclusión de que la solución óptima para cumplir dicha necesidad sería cubierta a través del procesamiento de anotaciones.

El primer paso es, por tanto, como en cualquier nuevo desarrollo software, llevar a cabo un análisis de los requisitos que ha de cumplir el nuevo tipo anotación y realizar un diseño de dicho tipo anotación que se ajuste a dichos requisitos.

En función de las requisitos planteados, deberán decidirse todos los detalles pertinentes, como, por ejemplo, sobre qué elementos de código se aplicará el tipo anotación (`@Target`), cuál debería ser su política de retención (`@Retention`), si no tendrá elementos (anotación marcadora) o, si necesita elementos, qué elementos concretamente y de qué tipos serían los más apropiados.

El diseño de un tipo anotación podrá ser más o menos sencillo dependiendo del número y la complejidad de los requisitos a satisfacer. Sin embargo, en general, un principio de diseño que habría que tratar de seguir en lo posible es que el tipo anotación debería permanecer sencillo para alentar su uso por parte de los usuarios a los que vaya dirigido. Está claro que las anotaciones marcadoras (sin elementos) o las anotaciones sencillas (con un único elemento) son las más atractivas por la comodidad de su uso. No obstante, si los requisitos lo imponen, podría necesitarse definir un gran número de elementos, como ocurre en el caso del tipo anotación `@DataSourceDefinition`, con casi una veintena de elementos.

La principal prioridad a la hora de diseñar un tipo anotación será tratar de ajustarse siempre lo mejor posible a los requisitos. En nuestro caso, para mantener este primer ejemplo sencillo, vamos a diseñar un tipo anotación que podrá aplicarse a cualquier tipo de declaración y que simplemente servirá para imprimir un mensaje, así que únicamente incluirá un elemento de tipo `String` para el mensaje con un valor por defecto:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface ImprimirMensaje {
    String value() default ";Hola, mundo! (mensaje por defecto)";
}
```

10.4.2.- Implementación del Procesador de anotaciones.

Una vez tenemos listo el diseño del tipo anotación a procesar, el siguiente paso es implementar el procesador de anotaciones correspondiente. Para ello, en J2SE 1.5, como hemos visto, dispondremos de las facilidades dadas por la Mirror API y toda la infraestructura de **apt**.

El primer paso y más importante en la implementación de un procesador de anotaciones es decidir la estrategia a seguir para localizar las declaraciones a procesar. En el subapartado “Descubrimiento de declaraciones a procesar” ya vimos que había dos métodos fundamentales:

- 1) `Collection<TypeDeclaration> getSpecifiedTypeDeclarations()`
- 2) `Collection<Declaration> getDeclarationsAnnotatedWith(AnnotationTypeDeclaration a)`

Con `getSpecifiedTypeDeclarations` podía obtenerse la lista de tipos sobre los que se había invocado el procesamiento y navegar por toda la jerarquía de declaraciones buscando las declaraciones que estaban anotadas con el tipo anotación a procesar. Para facilitar esta navegación podíamos servirnos de los Visitor, especialmente de los `DeclarationScanner` para transportar la lógica a implementar en función del tipo de declaración a todos los elementos de la jerarquía de declaraciones.

Con `getDeclarationsAnnotatedWith` se obtenía directamente la lista de declaraciones anotadas con el tipo anotación a procesar, lo cual ahorra mucho trabajo. No obstante, este método devuelve las declaraciones de todos los tipos mezcladas y sin ningún orden garantizado, por lo que no podrá ser utilizado en los casos en los que el procesamiento de elementos deba mantener un cierto orden concreto o realizarse por bloques específicos.

En el caso de nuestro tipo anotación `@ImprimirMensaje`, no hemos establecido ningún requisito concreto sobre el orden en el que pueden imprimirse los mensajes, así que optaremos por utilizar `getDeclarationsAnnotatedWith` para recuperar las declaraciones anotadas de forma directa y las procesaremos en función de su tipo.

El código fuente completo y profusamente comentado del procesador de anotaciones para el tipo anotación `@ImprimirMensaje`, implementado en la clase `ImprimirMensajeProcessor` puede encontrarse en el material complementario que se acompaña junto con este manual, dentro del proyecto `anotaciones-proc-java5`.

No obstante, aunque no es posible incluir el código fuente completo del procesador por cuestiones de espacio, sí vamos a comentar algunas cuestiones de especial interés.

En esta primera implementación de un procesador de anotaciones vamos a utilizar la estrategia más sencilla posible: obtenemos directamente todas las declaraciones anotadas con el tipo anotación a procesar a través del método `getDeclarationsAnnotatedWith`, iteramos sobre ellas en un bucle `for` y las vamos procesando según sea su tipo en la rama de un `if` de gran tamaño. En este caso concreto, se ha incluido también una pequeña comprobación adicional que nos ayudará a poder mostrar información de las declaraciones de paquetes, siempre y cuando estos tengan al menos una declaración relacionada. Por tanto, la estructura sería la siguiente:

```
// ITERACIÓN SOBRE LAS DECLARACIONES ANOTADAS

// recuperamos la lista de declaraciones anotadas
// con el tipo anotación a procesar por este procesador
Collection<Declaration> declsAnotadas =
    this.env.getDeclarationsAnnotatedWith(this.tipoDeclAnotacionAProcesar);

// iteramos sobre las declaraciones anotadas para ir procesándolas
for (Declaration decl : declsAnotadas) {

    // COMPROBACIÓN PREVIA: DECLARACIONES DE PAQUETES NO MOSTRADAS

    // ... información de las declaraciones de paquete no mostradas hasta el momento ...

    // PROCESAMIENTO DE LA DECLARACIÓN SEGÚN SU TIPO

    if (decl instanceof EnumDeclaration) {

        // DECLARACIÓN DE TIPO ENUMERADO
        // (EnumDeclaration es una subinterfaz de ClassDeclaration)

        // down-cast
        EnumDeclaration enumDecl = (EnumDeclaration) decl;

        // recuperamos la anotación que queremos procesar
        ImprimirMensaje enumAnotacion =
            enumDecl.getAnnotation(this.claseAnotacionAProcesar);

        // imprimimos su información
        messenger.printNotice(    enumDecl.getPosition(), "ENUMERADO ANOTADO");
        messenger.printNotice("ENUMERADO: " + enumDecl.getQualifiedName());
        messenger.printNotice("MENSAJE: " + enumAnotacion.value());

    } else if (decl instanceof ClassDeclaration) {

        // DECLARACIÓN DE CLASE
        // ... procesamiento de la declaración de clase ...

    } else if (decl instanceof AnnotationTypeDeclaration) {

        // DECLARACIÓN DE TIPO ANOTACIÓN
        // ... procesamiento de la declaración de tipo anotación ...

    } else if (decl instanceof InterfaceDeclaration) {

        // DECLARACIÓN DE INTERFAZ
        // ... procesamiento de la declaración de interfaz ...

    } else if ... otros tipos de declaraciones ...
}
```

Nótese que, en el fragmento de código anterior, en el bloque ***if*** se interroga primero por la interfaz **EnumDeclaration** antes que por **ClassDeclaration**. Esto es así debido a que **EnumDeclaration** es una subinterfaz de **ClassDeclaration**, por lo que, si el orden del ***if*** fuera preguntar primero por **ClassDeclaration**, al ser esta una superinterfaz, **ClassDeclaration** ocultaría a **EnumDeclaration**.

Por este motivo, y para evitar que las superinterfaces oculten a sus subinterfaces, en los bloques ***if*** sobre tipos de declaraciones (o tipos, ya que ocurre lo mismo), tiene que hacerse teniendo en cuenta la jerarquía de relaciones de herencia entre las distintas interfaces definidas. Para las declaraciones y los tipos, se sugiere utilizar el siguiente orden, respectivamente:

Construcción de bloques condicionales - Orden recomendado de Declaraciones y Tipos	
Declaraciones	Tipos
EnumDeclaration ClassDeclaration AnnotationTypeDeclaration InterfaceDeclaration TypeDeclaration EnumConstantDeclaration FieldDeclaration AnnotationTypeElementDeclaration MethodDeclaration ConstructorDeclaration ExecutableDeclaration MemberDeclaration ParameterDeclaration TypeParameterDeclaration	EnumType ClassType AnnotationType InterfaceType DeclaredType ArrayType TypeVariable ReferenceType WildcardType PrimitiveType VoidType

NOTA: Algunas tipos de declaraciones más abstractas por ser entidades de más alto nivel, como **TypeDeclaration** o **ExecutableDeclaration** podrían no ser necesarias y no aparecer si en lo que estamos interesados son en los elementos terminales de sus subinterfaces, que son los que realmente modelan elementos de código del lenguaje Java. Será muy común, por ejemplo, que nos interese realizar una lógica diferenciada para procesar las declaraciones de interfaces y clases, por lo que **TypeDeclaration** no aparecería en el ***if***.

Construcciones condicionales de Declaraciones	Construcciones condicionales de Tipos
<pre>if (decl instanceof EnumDeclaration) { // ...procesado de la declaración de clase... } else if (decl instanceof InterfaceDeclaration) { // ...procesado de la declaración de interfaz... } else if ...</pre>	<pre>if (type instanceof EnumType) { // ...procesado del tipo clase... } else if (type instanceof InterfaceType) { // ...procesado del tipo interfaz ... } else if ...</pre>

Por supuesto, este orden es sólo el recomendado y puede emplearse otro, siempre y cuando se respeten las relaciones de herencia entre los diferentes tipos de elementos de código. Además, si interesaría que se realice la lógica de todos los tipos a los que sea aplicable un objeto, por ejemplo si estamos realizando algún tipo de estadísticas, no habría más que convertir el ***if*** grande (***if...*** ***else if...*** ***else if...***) en distintos ***if*** separados (***if...*** ***if...*** ***if...***) para que, por ejemplo, una declaración de un tipo enumerado entrara tanto por la rama de **EnumDeclaration** como por la rama de **ClassDeclaration**.

Para ahorrarnos tener que construir esos bloques **if** tan grandes, así como el down-cast implícito que conlleva cada una de sus ramas, son algunos de los principales los motivos por los que podemos utilizar Visitor. Implementar la lógica de procesamiento en los métodos de los Visitor en lugar de en las ramas de los bloques **if** supone también una importante mejora estructural del código.

Una vez se ha elegido la estrategia de descubrimiento de las declaraciones anotadas a procesar y se ha montado la estructura condicional que nos permite procesar cada tipo de declaración de forma específica según nuestras necesidades, sólo queda llenar las ramas del bloque **if** con el tratamiento particular de cada tipo de declaración.

En este caso, por ser nuestro primer procesador de anotaciones y por tener el tipo anotación **@ImprimirMensajes** unos requisitos tan sencillos como simplemente imprimir los mensajes por la salida del compilador sin ningún orden específico, el procesamiento de todos los tipos de declaración son muy sencillos y muy parecidos: simplemente mostramos por la salida del compilador una serie de mensajes informativos sobre la posición en el código fuente de la declaración, su nombre y su mensaje asociado:

```
if (decl instanceof EnumDeclaration) {  
  
    // DECLARACIÓN DE TIPO ENUMERADO  
    // (EnumDeclaration es una subinterfaz de ClassDeclaration)  
  
    // down-cast  
    EnumDeclaration enumDecl = (EnumDeclaration) decl;  
  
    // recuperamos la anotación que queremos procesar  
    ImprimirMensaje enumAnotacion =  
        enumDecl.getAnnotation(this.claseAnotacionAProcesar);  
  
    // imprimimos su información  
    messenger.printNotice(enumDecl.getPosition(), "ENUMERADO ANOTADO");  
    messenger.printNotice("ENUMERADO: " + enumDecl.getQualifiedName());  
    messenger.printNotice("MENSAJE: " + enumAnotacion.value());  
  
} else if (decl instanceof ClassDeclaration) ...
```

Una vez completada la lógica de todos los tipos de declaración, podemos dar por concluida la implementación de **ImprimirMensajeProcessor**, el procesador de anotaciones del tipo anotación **@ImprimirMensaje**.

No obstante, antes de pasar a la siguiente fase y de que el tipo anotación y su procesador correspondiente sean usables por usuarios externos y se puedan invocar sobre ellos el proceso de procesamiento de anotaciones, es necesario implementar también la factoría del procesador creando una clase que implemente la interfaz **AnnotationProcessorFactory**.

Así pues, para crear una factoría de procesadores de anotaciones que pueda servir a **apt** cuantas instancias necesite de nuestro procesador a lo largo del procesamiento de anotaciones, crearemos una clase que implementará los métodos la interfaz de las factorías de procesadores de anotaciones: **AnnotationProcessorFactory**.

Las factorías de procesadores no suelen ser muy complicadas: se devuelve el nombre canónico del tipo anotación soportado por el procesador en **supportedAnnotationTypes()**, se devuelven las opciones del procesador en **supportedOptions()**, y se devuelve una nueva instancia del procesador de anotaciones llamando a su constructor en **getProcessorFor**.

Por ejemplo, para nuestro procesador del tipo anotación `@ImprimirMensaje`, el código de la clase factoría, que incluimos completo por no ser excesivamente largo, sería el siguiente:

```
public class ImprimirMensajeProcessorFactory implements AnnotationProcessorFactory {

    // -----
    // MÉTODOS DE APLICACIÓN
    // -----

    public Collection<String> supportedAnnotationTypes() {

        // nombres cualificados de los tipos anotación
        // soportados por este procesador
        String[] tiposAnotacionSoportados =
            { ImprimirMensaje.class.getCanonicalName() };

        // devolvemos los tipos soportados como una lista
        return Arrays.asList(tiposAnotacionSoportados);
    }

    public Collection<String> supportedOptions() {

        // devolvemos un conjunto vacío, ya que este
        // procesador no soporta ninguna opción específica
        return Collections.emptySet();
    }

    public AnnotationProcessor getProcessorFor(
        Set<AnnotationTypeDeclaration> atds,
        AnnotationProcessorEnvironment env) {

        // si el conjunto de tipos anotación viene vacío y
        // este no es un procesador universal (que soporta "*"),
        // es una buena práctica devolver el procesador identidad
        if (atds.isEmpty()) return AnnotationProcessors.NO_OP;

        // devolvemos una instancia del procesador correspondiente
        return new ImprimirMensajeProcessor(atds, env);
    }
}
```

NOTA: Nótese que en el método `getProcessorFor` nos hemos adherido a una buena práctica descrita en la documentación de la Mirror API que dice que, cuando se invoca dicho método con un conjunto vacío de declaraciones de tipos anotación a procesar, se considera recomendable, siempre que no vaya en contra de nuestras necesidades, devolver el procesador identidad, que no realiza ninguna operación de procesamiento (de ahí su nombre `NO_OP`).

10.4.3.- Anotación de elementos de código.

Una vez hemos completado el diseño del tipo anotación y la implementación de su correspondiente procesador de anotaciones, ya es posible utilizar el tipo anotación en el código fuente de los usuarios potenciales a los que vaya destinado.

Para nuestro ejemplo, se han creado varios ficheros de código fuente para demostrar todas las formas posibles de uso a la hora de utilizar una anotación. El paquete donde se han ubicado el tipo anotación, su procesador y su factoría, tiene un subpaquete `elementosAnotados` que contiene la definición de un paquete (`package-info.java`), una clase, una interfaz y un tipo anotación, para mostrar por la salida del procesador al menos una declaración de cada tipo:

```
@ImprimirMensaje("Definicion de la clase.")
public class ImprimirMensajeClaseAnotada {

    // =====
    // VARIABLES DE CLASE
    // =====

    @ImprimirMensaje("Variable de clase tipo String.")
    protected static String variableClase = "Valor de la Variable de clase.";

    // =====
    // VARIABLES DE INSTANCIA
    // =====

    @ImprimirMensaje("Variable de instancia tipo Float.")
    protected Float variableInstanciaFloat = 666.666f;

    // =====
    // TIPO ENUMERADO
    // =====

    @ImprimirMensaje("Tipo Enumerado interno")
    public enum ImprimirMensajeClaseAnotadaEnum {

        @ImprimirMensaje("Constante Tipo Enumerado 1") valorEnum1,
        @ImprimirMensaje("Constante Tipo Enumerado 2") valorEnum2;
    }

    // =====
    // CONSTRUCTORES
    // =====

    @ImprimirMensaje("Constructor de la clase.")
    public ImprimirMensajeClaseAnotada() { }

    // =====
    // MÉTODOS DE APLICACIÓN
    // =====

    @ImprimirMensaje("Método imprimirMensaje")
    public void imprimirMensaje(
        @ImprimirMensaje("Parámetro del método imprimirMensaje") String mensaje) {

        @ImprimirMensaje("Variable local. NO SE SOPORTA EL "
            + "PROCESAMIENTO DE ANOTACIONES SOBRE VARIABLES LOCALES. "
            + "Un procesador J2SE 1.5 nunca llegará a ver esta anotación.")
        String variableLocal = "Mensaje: ";
        System.out.println(variableLocal + mensaje);
    }
}
```

El código de clase anotada definida, como puede observarse, está definido para crear al menos una declaración de todas las que es posible definir dentro de una clase. Los tipos de declaraciones restantes se han realizado entre los demás ficheros.

Así pues, nótese como en el código de la clase anotada de ejemplo se incluyen anotaciones sobre elementos de código que no se suelen utilizar muy frecuentemente como, por ejemplo: anotaciones sobre un tipo enumerado y sus constantes, una anotación sobre el parámetro de un método o incluso una anotación sobre una variable local. Eso sí, la anotación sobre la variable local se incluye meramente a efectos ilustrativos, ya que, como sabemos, el procesamiento de anotaciones sobre variables locales no será introducido hasta Java SE 8.

10.4.4.- Invocación de apt y obtención de resultados.

Teniendo ya el tipo anotación y su procesador implementados, y el código cliente de ejemplo debidamente anotado, ya estamos en condiciones de realizar nuestra primera invocación a apt para llevar a cabo el procesamiento de anotaciones. Aunque es posible invocar **apt** de múltiples formas, en este apartado utilizaremos la línea de comandos directamente para ilustrar su uso según la documentación oficial.

En nuestro caso, para realizar nuestro procesamiento de anotaciones, ejecutaremos desde el directorio raíz del proyecto Eclipse llamado **anotaciones-proc-java5** el siguiente comando:

```
apt.exe -version ^
-XListAnnotationTypes -XListDeclarations -XPrintAptRounds -XPrintFactoryInfo ^
-cp ".\target\classes;.\lib\mirror-api-1.0.0.jar" ^
-s .\src\main\java-generated ^
-d .\target\classes ^
-factory anotaciones.procesamiento.java5.ejemplos.imprimirMensaje.ImprimirMensajeProcessorFactory ^
.\src\main\java\anotaciones\procesamiento\java5\ejemplos\imprimirMensaje\elementosAnotados\*.java
```

Por ser el primer ejemplo de procesamiento de anotaciones, se han incluido en el comando de invocación a **apt** todas las opciones no estándar de depuración (las que empiezan por **-x**). De esta forma, el lector podrá constatar y realizar un seguimiento por sí mismo del modo de operación que sigue **apt** y que se encuentra explicado con detalle en el subapartado “Funcionamiento de **apt**”, como, por ejemplo, el número de la ronda, la lista de declaraciones especificadas y la lista de declaraciones incluidas, los tipos anotación detectado para ser procesados, las factorías disponibles encontradas en el proceso de descubrimiento y otra información que puede ser de gran relevancia a la hora de depurar el comportamiento de nuestro procesador de anotaciones.

Además de las opciones anteriores, puede también añadirse la opción **-verbose** para una salida aún más detallada del proceso, esta vez incluyendo también información sobre las acciones de análisis y compilación de ficheros de código fuente.

Una vez invocado **apt**, se realizará el procesamiento de anotaciones tal y como habíamos implementado en el procesador para el tipo anotación **@ImprimirMensaje**, obteniendo por la salida del compilador la información de los elementos de código anotados y sus mensajes:

```
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\imprimirMensaje\elementosAnotados\pack-
age-info.java:5: Note: PAQUETE ANOTADO
@ImprimirMensaje("Paquete java5.ejemplos.imprimirMensaje.elementosAnotados")
^
Note: PAQUETE: anotaciones.procesamiento.java5.ejemplos.imprimirMensaje.elementosAnotados
Note: MENSAJE: Paquete java5.ejemplos.imprimirMensaje.elementosAnotados
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\imprimirMensaje\elementosAnotados\Impr-
imirMensajeTipoAnotacionAnotado.java:11: Note: TIPO ANOTACIÓN ANOTADO
public @interface ImprimirMensajeTipoAnotacionAnotado {
    ^
Note: TIPO ANOTACIÓN:
anotaciones.procesamiento.java5.ejemplos.imprimirMensaje.elementosAnotados.ImprimirMensajeTipoAnota-
cionAnotado
Note: MENSAJE: Tipo anotación anotado.
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\imprimirMensaje\elementosAnotados\Impr-
imirMensajeClaseAnotada.java:9: Note: CLASE ANOTADA
public class ImprimirMensajeClaseAnotada {
    ^
Note: CLASE:
anotaciones.procesamiento.java5.ejemplos.imprimirMensaje.elementosAnotados.ImprimirMensajeClaseAnot-
ada
Note: MENSAJE: Definicion de la clase.
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\imprimirMensaje\elementosAnotados\Impr-
imirMensajeClaseAnotada.java:51: Note: PARAMETRO ANOTADO
    @ImprimirMensaje("Parámetro del método imprimirMensaje") String mensaje) {
    ^
Note: PARAMETRO: java.lang.String mensaje
Note: MENSAJE: Parámetro del método imprimirMensaje
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\imprimirMensaje\elementosAnotados\Impr-
imirMensajeClaseAnotada.java:30: Note: ENUMERADO ANOTADO
    public enum ImprimirMensajeClaseAnotadaEnum {
        ^
Note: ENUMERADO:
anotaciones.procesamiento.java5.ejemplos.imprimirMensaje.elementosAnotados.ImprimirMensajeClaseAnot-
ada.ImprimirMensajeClaseAnotadaEnum
Note: MENSAJE: Tipo Enumerado interno
...
    algunos mensajes más ...
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\imprimirMensaje\elementosAnotados\Impr-
imirMensajeClaseAnotada.java:23: Note: CAMPO ANOTADO
    protected Float variableInstanciaFloat = 666.666f;
    ^
Note: CAMPO: variableInstanciaFloat
Note: MENSAJE: Variable de instancia tipo Float.
```

Con esto, hemos completado todo el proceso de creación y diseño de un tipo anotación, implementación de su procesador de anotaciones, uso del tipo anotación diseñado por parte del código cliente y, finalmente, su correspondiente procesamiento y obtención de resultados.

En los siguientes apartados expondremos más ejemplos de tipos anotación con requisitos más complejos y que cubren la diversidad de aplicaciones más habituales cuando se trabaja con procesamiento de anotaciones, como son la validación o la generación de código.

10.5.- Ejemplos: Procesadores J2SE 1.5.

En este apartado se proponen varios ejemplos para profundizar en todas las etapas de implementación del procesamiento de anotaciones en J2SE 1.5. Se ofrecen ejemplos para las principales aplicaciones del procesamiento de anotaciones: (1) navegación e información sobre el código fuente, (2) validación y (3) generación de ficheros. Para cada una de estas aplicaciones se ofrece un ejemplo básico y otro más avanzado para poder profundizar algo más en el tema.

10.5.1.- Navegación básica: @ImprimirMensaje.

Se trata del ejemplo que hemos utilizado en el apartado anterior para explicar paso a paso las diferentes fases por las que pasa todo el proceso de procesamiento de anotaciones.

Requisitos:

Diseñar un tipo anotación que permita imprimir por la salida del compilador un mensaje textual asociado a cualquier tipo de elemento del lenguaje Java.

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface ImprimirMensaje {

    String value() default "¡Hola, mundo! (mensaje por defecto)";
}
```

Implementación:

Proyecto: **anotaciones-proc-java5**.
Paquete: **anotaciones.procesamiento.java5.ejemplos.imprimirMensaje**.

Clases anotadas:

Las clases anotadas, al ser muchas y para que no se mezclaran con las clases que implementan el procesamiento, se han alojado en el subpaquete llamado **elementosAnotados**.

Invocación:

```
apt.exe -version ^
-XListAnnotationTypes -XListDeclarations -XPrintAptRounds -XPrintFactoryInfo ^
-cp ".\target\classes;.\lib\mirror-api-1.0.0.jar" ^
-s .\src\main\java-generated ^
-d .\target\classes ^
-factory anotaciones.procesamiento.java5.ejemplos.imprimirMensaje.ImprimirMensajeProcessorFactory ^
.\src\main\java\anotaciones\procesamiento\java5\ejemplos\imprimirMensaje\elementosAnotados\*.java
```

Resultados:

Una vez invocado correctamente **apt** y realizado el procesamiento de anotaciones por parte del procesador del tipo anotación, obtenemos por la salida del compilador la información de los elementos de código anotados y sus mensajes, como hemos visto en el apartado anterior.

10.5.2.- Navegación avanzada: @Nota.

En este ejemplo vamos a hacer un uso más avanzado de la navegación por la jerarquía de declaraciones usando las facilidades dadas por la Mirror API para utilizar el patrón Visitor. Se utilizará un **SourceOrderDeclarationScanner** para recorrer dichas declaraciones en un orden lo más próximo posible al orden original en el que aparecen en el código fuente.

Además, introduciremos más requisitos de funcionalidad sobre el tipo anotación a diseñar, así como otras características avanzadas, como opciones para el procesador o la utilización de un listener para ser notificados al final de la ronda de procesamiento.

Requisitos:

Diseñar un tipo anotación que imprima por la salida del compilador, en el orden más similar posible al código fuente, notas de texto asociadas a cualquier tipo de elemento del lenguaje Java.

Para ofrecer la posibilidad de extender la funcionalidad del tipo anotación, estas notas se podrán calificar en tipos: **NOTA** (por defecto), **DOCUMENTACION**, **TAREA**, **PENDIENTE**, **AVISO** y **ERROR**. Las notas de tipo **AVISO** y **ERROR** generarán un warning y un error respectivamente a la salida del compilador. Esto no tiene mucha utilidad práctica, si no que se incluye a efectos ilustrativos. Se deberán ofrecer opciones para incluir o excluir específicamente una lista de cualquiera de estos tipos de notas. Por defecto: incluir **TODO**, excluir **NADA**.

Además, se podrá establecer un valor de severidad de la nota respectiva relacionada con la importancia de su contenido: **NORMAL** (por defecto), **LEVE**, **GRAVE** y **FATAL**. Se deberán ofrecer opciones para incluir o excluir específicamente una lista de cualquiera de estas severidades. Por defecto: incluir **TODO**, excluir **NADA**.

Para permitir ordenar, si se desea, las diferentes notas atendiendo a diferentes criterios, el tipo anotación incluirá un campo de prioridad de tipo entero, donde la prioridad irá, en principio, de mayor a menor, por lo que la mayor prioridad vendrá dada por **Integer.MAX_VALUE**, la menor por **Integer.MIN_VALUE** y el valor por defecto será **0**. No obstante, deberá poderse configurar mediante las opciones pertinentes los rangos que queremos seleccionar (por defecto: el rango completo), así como si se ordenará de forma ascendente (por defecto) o descendente.

Opciones soportadas:

Opciones soportadas por el procesador del tipo anotación @Nota		
Opción	Valores válidos	Por defecto
-Atipo.nota.incluir -Atipo.nota.excluir	Lista de tipos de notas separados por comas. Valores especiales: TODO y NADA . Si se encuentra alguno de estos valores especiales, se ignoran los demás.	Incluir TODO . Excluir NADA .
-Aseveridad.incluir -Aseveridad.excluir	Lista de tipos de severidad separados por comas. Valores especiales: TODO y NADA . Si se encuentra alguno de estos valores especiales, se ignoran los demás.	Incluir TODO . Excluir NADA .
-Aprioridad.minima -Aprioridad.maxima	Rango de prioridad mínima y máxima. Las notas que queden fuera de este rango no serán procesadas.	Mín: Integer.MIN_VALUE Máx: Integer.MAX_VALUE
-Aorden.tipo	Tipo de ordenación de las notas de entre las disponibles: CODIGO (por orden de aparición en código fuente, por defecto), TIPO , SEVERIDAD , PRIORIDAD .	CODIGO
-Aorden.direccion	Dirección de la ordenación de las notas: ASC (ascendente, por defecto) y DESC (descendente).	ASC

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Nota {

    // =====
    // ELEMENTOS
    // =====

    String texto() default "n/a";
    TipoNota tipo() default TipoNota.NOTA;
    SeveridadNota severidad() default SeveridadNota.NORMAL;
    int prioridad() default 0;

    // =====
    // TIPOS ENUMERADOS ESTÁTICOS
    // =====

    public static enum TipoNota { NOTA, DOCUMENTACION, TAREA, PENDIENTE, AVISO, ERROR };
    public static enum SeveridadNota { NORMAL, LEVE, GRAVE, FATAL };

    public static enum TipoOrden { CODIGO, TIPO, SEVERIDAD, PRIORIDAD };
    public static enum DireccionOrden { ASC, DESC };
}
```

Implementación:

Proyecto: **anotaciones-proc-java5**.

Paquete: **anotaciones.procesamiento.java5.ejemplos.nota**.

Clases anotadas:

Se incluye una clase anotada llamada **NotaClaseAnotada** de ejemplo en el mismo paquete.

Invocación:

La invocación de este ejemplo incluye, además de las opciones no estándar de depuración, que nos pueden venir bien para ver detalles del procesamiento en sí, todas las opciones soportadas por el procesador del tipo anotación **@Nota**, por lo que la invocación sale bastante aparatoso:

```
apt.exe -version ^
-XListAnnotationTypes -XListDeclarations -XPrintAptRounds -XPrintFactoryInfo ^
-cp ".\target\classes;.\lib\mirror-api-1.0.0.jar" ^
-s .\src\main\java-generated ^
-d .\target\classes ^
-factory anotaciones.procesamiento.java5.ejemplos.nota.NotaProcessorFactory ^
-Atipo.nota.incluir=NOTA,DOCUMENTACION,AVISO,TAREA,PENDIENTE,ERROR ^
-Atipo.nota.excluir=NADA ^
-Aseveridad.incluir=NORMAL,LEVE,GRAVE,FATAL ^
-Aseveridad.excluir=NADA ^
-Aprioridad.minima=-1234 ^
-Aprioridad.maxima=1234 ^
-Aorden.tipo=CODIGO ^
-Aorden.direccion=ASC ^
.\src\main\java\anotaciones\procesamiento\java5\ejemplos\nota\*.java
```

Resultados:

Una vez invocado **apt** y realizado el procesamiento, se nos presenta por la salida del compilador las declaraciones que se ajusten a los tipos, severidades y prioridades especificadas, ordenadas según el orden especificado a través de las opciones de configuración del procesador.

Para el procesador de este ejemplo, se han añadido líneas de información adicionales que informan de las etapas de procesamiento concretas que va ejecutando el procesador: registro del listener, procesamiento de las opciones de configuración, recuperación de las declaraciones de los tipos especificados y, finalmente, el procesamiento, ordenación y escritura de resultados.

Para la invocación dada, se obtienen los siguientes resultados:

```
Note: procesador: anotaciones.procesamiento.java5.ejemplos.nota.NotaProcessor
Note: procesador: iniciando procesador...
Note: procesador: listener: registrando listener...
Note: procesador: listener: listener registrado.
Note: procesador: opciones: parseando opciones...
Note: tiposNotasAProcesar = [NOTA, DOCUMENTACION, TAREA, PENDIENTE, AVISO, ERROR]
Note: severidadesAProcesar = [NORMAL, LEVE, GRAVE, FATAL]
Note: prioridadMinimaAProcesar = -1234
Note: prioridadMaximaAProcesar = 1234
Note: tipoOrdenAProcesar = CODIGO
Note: direccionOrdenAProcesar = ASC
Note: procesador: opciones: opciones parseadas.
Note: procesador: tipos especificados: recuperando tipos especificados...
Note: procesador: tipos especificados: 31 tipos especificados.
Note: procesador: procesamiento: iniciando procesamiento...
Note: procesador: procesamiento: procesamiento completado.
Note: procesador: listener: ronda completada: [final round: true, error raised: false, source files
created: false, class files created: false]
Note: procesador: ordenacion: ordenando resultados...
Note: procesador: ordenacion: resultados ordenados.
Note: procesador: resultados: escribiendo resultados...
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\nota\package-info.java:1: Note:
@Nota(texto = "Definición del paquete anotaciones.procesamiento.java5.ejemplos.nota",
^
Note: @Nota: Texto: Definición del paquete anotaciones.procesamiento.java5.ejemplos.nota
    Tipo: DOCUMENTACION, Severidad: NORMAL, Prioridad: 0
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\nota\NotaClaseAnotada.java:10: Note:
public class NotaClaseAnotada {
    ^
Note: @Nota: Texto: Definición de la clase NotaClaseAnotada
    Tipo: NOTA, Severidad: NORMAL, Prioridad: 1
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\nota\NotaClaseAnotada.java:20:
warning:
    protected static String variableClase = "Valor de la Variable de clase.";
    ^
warning: @Nota: Texto: Variable de clase de NotaClaseAnotada - AVISO
    Tipo: AVISO, Severidad: GRAVE, Prioridad: 111
...
... algunas notas más ...

D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\nota\NotaClaseAnotada.java:77: Note:
    prioridad = 888) String mensaje) {
    ^
Note: @Nota: Texto: Parámetro "mensaje" del método imprimirMensaje de la clase NotaClaseAnotada
    Tipo: DOCUMENTACION, Severidad: NORMAL, Prioridad: 888
Note: procesador: resultados: resultados escritos.
Note: procesador: anotaciones.procesamiento.java5.ejemplos.nota.NotaProcessor finalizado.
```

Comentarios:

En la implementación de este ejemplo avanzado de navegación por las jerarquías de declaraciones se ha tratado de ceñirse a las mejores prácticas recomendadas de programación.

Primeramente, se han definido tipos enumerados para modelar las características únicas del tipo anotación `@Nota`, ya que estos favorecen la posibilidad de incluir nuevos valores.

Además, se han definido las cadenas que modelan las opciones del procesador como constantes de la clase factoría. Podrían haberse definido igualmente en la clase del procesador, pero se han definido en la factoría para no aglomerar más código en el procesador.

```
public class NotaProcessorFactory implements AnnotationProcessorFactory {

    // =====
    // VARIABLES DE CLASE
    // =====

    // OPCIONES DEL PROCESADOR

    public static final String OPCION_TIPO_NOTA_INCLUIR = "-Atipo.nota.incluir";
    public static final String OPCION_TIPO_NOTA_EXCLUIR = "-Atipo.nota.excluir";

    public static final String OPCION_SEVERIDAD_INCLUIR = "-Aseveridad.incluir";
    public static final String OPCION_SEVERIDAD_EXCLUIR = "-Aseveridad.excluir";

    public static final String OPCION_PRIORIDAD_MINIMA = "-Aprioridad.minima";
    public static final String OPCION_PRIORIDAD_MAXIMA = "-Aprioridad.maxima";

    public static final String OPCION_ORDEN_TIPO      = "-Aorden.tipo";
    public static final String OPCION_ORDEN_DIRECCION = "-Aorden.direccion";

    // OTRAS CONSTANTES

    public static final String TODO   = "TODO";
    public static final String NADA   = "NADA";

    // =====
    // MÉTODOS DE APLICACIÓN
    // =====

    public Collection<String> supportedAnnotationTypes() { ... }

    public Collection<String> supportedOptions() {

        // lista de opciones soportadas por el procesador
        String[] opcionesSoportadas =
            {   OPCION_TIPO_NOTA_INCLUIR, OPCION_TIPO_NOTA_EXCLUIR,
                OPCION_SEVERIDAD_INCLUIR, OPCION_SEVERIDAD_EXCLUIR,
                OPCION_PRIORIDAD_MINIMA, OPCION_PRIORIDAD_MAXIMA,
                OPCION_ORDEN_TIPO, OPCION_ORDEN_DIRECCION     };

        // lista de opciones soportadas
        return Arrays.asList(opcionesSoportadas);
    }

    public AnnotationProcessor getProcessorFor(
        Set<AnnotationTypeDeclaration> atds, AnnotationProcessorEnvironment env) { ... }
}
```

Esta estrategia siempre está recomendada porque facilita el mantenimiento y refactorización del código fuente de forma sencilla en caso de que hubiera que efectuar cualquier tipo de modificaciones sobre los requisitos a implementar.

Lo siguiente es comentar brevemente la estructura del procesamiento implementado para este procesador. Se ha elegido registrar un listener para presentar los resultados al final de la ronda de procesamiento y procesar las declaraciones de los tipos especificados con una instancia de **SourceOrderDeclarationScanner**. Se han tenido en cuenta las declaraciones de paquetes.

Aquí sigue un esquema simplificado del método **process** del código fuente del ejemplo:

```

public void process() {

    // REGISTRO DEL PROCESADOR COMO LISTENER
    this.env.addListener(this);

    // OPCIONES DEL PROCESADOR
    this.parsearYProcesarOpcionesProcesador();

    // CREACIÓN DE FILTRO DE DECLARACIONES ELEGIBLES PARA EL RESULTADO
    filtroResultado = new DeclarationFilter() { ... };

    // LISTA DE DECLARACIONES RESULTADO
    declsResultado = new ArrayList<Declaration>();

    // PROCESAMIENTO DE LAS DECLARACIONES DE PAQUETES

    // lista de declaraciones de paquetes encontradas hasta el momento
    pqDeclsEncontradas = new ArrayList<PackageDeclaration>();

    // PROCESAMIENTO DE ANOTACIONES DE LOS TIPOS ESPECIFICADOS

    Collection<TypeDeclaration> tiposEspecificados = this.env.getSpecifiedTypeDeclarations();

    // instanciamos un Visor de tipo SourceOrderDeclarationScanner para
    // vanegar por la jerarquía de declaraciones en un orden lo más parecido
    // posible al del código fuente original, utilizando como Visor de
    // pre-procesamiento una instancia de nuestra clase interna NotaDeclarationVisitor
    DeclarationVisitor notaSourceOrderDeclarationScanner =
        DeclarationVisitors.getSourceOrderDeclarationScanner(
            new NotaDeclarationVisitor(), // Visor de pre-procesamiento
            DeclarationVisitors.NO_OP); // Visor de post-procesamiento

    // enviamos el scanner a todas las declaraciones de los tipos especificados
    for (TypeDeclaration typeDecl : tiposEspecificados) {

        // enviamos el scanner a la declaración de tipo, con lo cual se visitarán,
        // al ser un DeclarationScanner, no sólo las propias definiciones de tipo,
        // si no también todas las subdeclaraciones de dichos tipos
        typeDecl.accept(notaSourceOrderDeclarationScanner);
    }

    messenger.printNotice("procesador: procesamiento: procesamiento completado.");
}

// NOTA: No se hace nada más en process(). Todo el procesamiento se acabará
// haciendo en el método auxiliar procesarDeclaracion(Declaration decl), que
// será el que llame el scanner para todas las declaraciones y subdeclaraciones
// de los tipos especificados. En procesarDeclaracion guardaremos la información
// de las anotaciones procesadas y, al final de la ronda, en el entorno del
// listener, en su método roundComplete, presentaremos finalmente todos los
// resultados filtrados y ordenados según las opciones del procesador.

} // process

```

Por tanto, es en el método `procesarDeclaracion` donde se hace el procesamiento de las declaraciones, que en este caso es simplemente pasarlas por el filtro de resultados para ver si son elegibles para ser incluidas en el resultado. Realmente es muy sencillo una vez que se tiene construido el filtro. Es más aparatoso incluso el código inicial que se incluye para tratar el caso especial para capturar las declaraciones de paquetes.

```
private void procesarDeclaracion(Declaration decl) {

    // CASO ESPECIAL PARA CAPTURAR LAS DECLARACIONES DE PAQUETES

    // antes de comprobar la declaración como tal,
    // vamos a comprobar la declaración de paquete (si es posible)
    if (decl instanceof TypeDeclaration) {

        // down-cast
        PackageDeclaration pqDecl = ((TypeDeclaration) decl).getPackage();

        // comprobamos si la declaración de paquete
        // debe incluirse en el resultado
        if (this.filtroResultado.matches(pqDecl)) {

            // incluimos la declaración de paquete en la lista resultado
            // y en la lista de declaraciones de paquetes encontradas
            // si dicha declaración no lo estaba ya de antes
            if (!pqDeclsEncontradas.contains(pqDecl)) {

                declsResultado.add(pqDecl);
                pqDeclsEncontradas.add(pqDecl);
            }
        }
    }

    // PROCESAMIENTO DE DECLARACIONES

    // comprobamos si la declaración debe incluirse en el resultado
    if (this.filtroResultado.matches(decl)) {

        // incluimos la declaración en la lista resultado
        declsResultado.add(decl);
    }
}
```

Finalmente, tras completarse la ronda de procesamiento, al haber registrado como listener el propio procesador, se invocará finalmente el método `roundComplete` definido en su cuerpo.

Y es que, para ilustrar un posible uso de los listeners, en este ejemplo se ha optado por escribir los resultados en el método manejador del evento `roundComplete` del procesador.

NOTA: Teniendo en cuenta que tenemos las declaraciones resultado en una colección, ya filtradas y ordenadas según los criterios especificados en las opciones del procesador, podríamos ampliar fácilmente la funcionalidad de nuestro procesador de anotaciones añadiendo como requisito, por ejemplo, la funcionalidad de volcar dicha información a un fichero de texto (usando el método `createTextFile` la interfaz `Filer` del entorno de procesamiento) o incluso alimentar la creación de informes a ficheros PDF, HTML, XLS, CSV, XML, etcétera, mediante la utilización de alguna API Java de informes (reporting) como, por ejemplo, [JasperReports](#). Cualquiera de estas posibilidades, se deja como ejercicio práctico para el lector.

En el método **roundComplete**, tras ordenar los resultados en función del criterio dado por las opciones del procesador, se escribe la información a la salida del compilador. Teniendo en cuenta además los requisitos que pedían que se arrojara un warning en caso de que el tipo de nota fuera **AVISO** y un error de compilación en el caso de que la nota fuera de tipo **ERROR**.

Esto último hace que podamos generar warnings y errores de compilación a placer estableciendo los valores oportunos en las anotaciones. Hacer esto no tiene mucho sentido en la práctica, pero se ha incluido e efectos ilustrativos de una de las capacidades más interesantes de los procesadores: la validación, funcionalidad esta a la que le dedicaremos los siguientes ejemplos.

```
public void roundComplete(RoundCompleteEvent event) {  
    // ORDENAMOS LOS RESULTADOS SEGÚN EL ORDEN CONFIGURADO  
  
    // ordenamos los resultados antes de mostrarlos  
    this.ordenarResultados();  
  
    // MOSTRAMOS LOS RESULTADOS POR LA SALIDA DEL COMPILOADOR  
  
    // una vez ordenamos, sólo hemos de mostrar los resultados  
    // por la salida del compilador  
    for (Declaration decl : this.declsResultado) {  
  
        // obtenemos la anotación @Nota correspondiente a la declaración  
        Nota anotacionNota = decl.getAnnotation(this.claseAnotacionAProcesar);  
  
        // construimos la representación del la información de la anotación  
        String anotacionNotaInfo = "@Nota: Texto: " + anotacionNota.texto() + "\n"  
            + "\t" + "Tipo: " + anotacionNota.tipo()  
            + ", Severidad: " + anotacionNota.severidad()  
            + ", Prioridad: " + anotacionNota.prioridad();  
  
        // imprimimos las declaraciones de las anotaciones resultado según su tipo  
        // si es un AVISO -> hay que generar un warning  
        // si es un ERROR -> hay que generar un error de compilación  
        // en cualquier otro caso -> se genera un mensaje meramente informativo  
        if (anotacionNota.tipo().equals(Nota.TipoNota.AVISO)) {  
  
            // emitimos un warning al compilador  
            messenger.printWarning(decl.getPosition(), "");  
            messenger.printWarning(anotacionNotaInfo);  
  
        } else if (anotacionNota.tipo().equals(Nota.TipoNota.ERROR)) {  
  
            // emitimos un error de compilación  
            messenger.printError(decl.getPosition(), "");  
            messenger.printError(anotacionNotaInfo);  
  
        } else {  
  
            // emitimos un mensaje mensaje informativo  
            messenger.printNotice(decl.getPosition(), "");  
            messenger.printNotice(anotacionNotaInfo);  
        }  
    }  
}
```

10.5.3.- Validación básica: @Contexto.

En el siguiente ejemplo vamos a introducir una de las funcionalidades más habituales para las que se utiliza el procesamiento de anotaciones: la **validación de código fuente**.

Al disponer, gracias a la Mirror API, de las facilidades para navegar por los elementos del código en tiempo de compilación, poder evaluar sus diferentes propiedades y emitir errores de compilación desde el entorno del procesador de anotaciones, estamos en condiciones de implementar procesadores de anotaciones que, además de realizar otras acciones, se encarguen de comprobar y validar que un cierto código fuente cumpla ciertas condiciones necesarias que interese asegurar para el correcto funcionamiento de las aplicaciones ya en tiempo de ejecución.

Requisitos:

Diseñar un tipo anotación aplicable sobre campos que permita comprobar si el tipo declarado de dicho campo pertenece a una implementación de una interfaz de contexto de una aplicación llamada, por ejemplo, **IContextoApp**.

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.FIELD)
public @interface Contexto { }
```

Implementación:

Proyecto: **anotaciones-proc-java5**.

Paquete: **anotaciones.procesamiento.java5.ejemplos.contexto**.

Clase anotada:

Se incluye la clase anotada **ContextoClaseAnotada** como ejemplo en el mismo paquete.

Invocación:

```
apt.exe -version ^
-XListAnnotationTypes -XListDeclarations -XPrintAptRounds -XPrintFactoryInfo ^
-cp ".\target\classes;.\lib\mirror-api-1.0.0.jar" ^
-s .\src\main\java-generated ^
-d .\target\classes ^
-factory anotaciones.procesamiento.java5.ejemplos.contexto.ContextoProcessorFactory ^
.\src\main\java\anotaciones\procesamiento\java5\ejemplos\contexto\*.java
```

Resultados:

Al invocar el procesamiento y procesarse la clase anotada se obtienen los resultados esperados: las declaraciones anotadas con `@Contexto` sobre campos cuyo tipo declarado o no implementa `IContextoApp` o es una interfaz provoca un error de compilación, mientras que los campos anotados con `@Contexto` sobre clases que implementan la interfaz `IContextoApp` no dan ningún problema. En concreto en la clase `ContextoClaseAnotada` se deben encontrar 7 declaraciones anotadas, como se ve en la salida del procesador, de las cuales 5 arrojan errores:

```
Note: procesador: anotaciones.procesamiento.java5.ejemplos.contexto.ContextoProcessor
Note: procesador: iniciando procesador...
Note: procesador: declaraciones anotadas: recuperando declaraciones anotadas...
Note: procesador: declaraciones anotadas: 7 declaraciones anotadas.
Note: procesador: procesamiento: iniciando procesamiento...
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\contexto\ContextoClaseAnotad
a.java:29: error: tipo declarado del campo variableInstanciaContextoInterfaz
(anotaciones.procesamiento.java5.ejemplos.contexto.IContextoApp) no implementa la
interfaz anotaciones.procesamiento.java5.ejemplos.contexto.IContextoApp
    protected static IContextoApp variableInstanciaContextoInterfaz;
                                         ^
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\contexto\ContextoClaseAnotad
a.java:40: error: tipo declarado del campo valorEnumNoContexto
(anotaciones.procesamiento.java5.ejemplos.contexto.ContextoClaseAnotada.TipoEnumerado) no
implementa la interfaz anotaciones.procesamiento.java5.ejemplos.contexto.IContextoApp
    @Contexto // ERROR: El tipo enumerado no implementa IContextoApp
                                         ^
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\contexto\ContextoClaseAnotad
a.java:32: error: tipo declarado del campo variableInstanciaNoContexto
(java.lang.Integer) no implementa la interfaz
anotaciones.procesamiento.java5.ejemplos.contexto.IContextoApp
    protected Integer variableInstanciaNoContexto = 1234;
                                         ^
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\contexto\ContextoClaseAnotad
a.java:19: error: tipo declarado del campo variableClaseNoContexto (java.lang.String) no
implementa la interfaz anotaciones.procesamiento.java5.ejemplos.contexto.IContextoApp
    protected static String variableClaseNoContexto;
                                         ^
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\contexto\ContextoClaseAnotad
a.java:16: error: tipo declarado del campo variableClaseContextoInterfaz
(anotaciones.procesamiento.java5.ejemplos.contexto.IContextoApp) no implementa la
interfaz anotaciones.procesamiento.java5.ejemplos.contexto.IContextoApp
    protected static IContextoApp variableClaseContextoInterfaz;
                                         ^
Note: procesador: procesamiento: procesamiento completado.
Note: procesador: anotaciones.procesamiento.java5.ejemplos.contexto.ContextoProcessor
finalizado.
5 errors
```

Comentarios:

El procesador de anotaciones para validar los requisitos del tipo anotación `@Contexto` es muy sencillo gracias a que la comprobación que se debe realizar es local a cada campo. Debido a esto, es posible, como se ve en la implementación del método `process` del procesador, simplemente recuperar todas las declaraciones anotadas a través de `getDeclarationsAnnotatedWith` y procesarlas iterando directamente sobre ellas sin ningún orden en particular.

Luego, en el procesamiento de cada declaración se recupera el tipo de la declaración del campo y se comprueba que no sea un tipo interfaz y sea assignable a `IContextoApp`, es decir: que sea una implementación de dicha interfaz.

A continuación se deja el contenido del método `process` del procesador, una vez suprimidas con respecto al código fuente completo original algunas líneas de código de mensajes informativos por restricciones de espacio:

```
public void process() {  
  
    // PROCESAMIENTO DE ANOTACIONES DE LAS DECLARACIONES ANOTADAS  
  
    // referencia a la declaración del tipo anotación  
    AnnotationTypeDeclaration atd =  
        (AnnotationTypeDeclaration) this.env.getTypeDeclaration(  
            this.claseAnotacionAProcesar.getCanonicalName());  
  
    // recuperamos la lista de tipos especificados a procesar por este procesador  
    Collection<Declaration> declAnotadas = this.env.getDeclarationsAnnotatedWith(atd);  
  
    // procesamos una a una y en un orden indeterminado (nos da igual el orden)  
    // todas las declaraciones anotadas con el tipo anotación a procesar  
    for (Declaration decl : declAnotadas) {  
  
        // PROCESAMIENTO DE LA DECLARACIÓN  
  
        // sabemos que la anotación sólo se puede usar sobre campos,  
        // así que podemos hacer un down-cast de forma segura  
        FieldDeclaration campoDecl = (FieldDeclaration) decl;  
  
        // obtenemos el tipo de la declaración del campo  
        TypeMirror campoTipo = campoDecl.getType();  
  
        // comprobamos si el tipo es assignable al tipo interfaz IContextoApp  
        boolean anotacionValida =  
            !(campoTipo instanceof InterfaceType)  
            && this.types.isAssignable(campoTipo, this.tipoIContextoApp);  
  
        // si no es una anotación válida, tenemos que emitir un error de compilación  
        if (!anotacionValida) {  
  
            messenger.printError(  
                decl.getPosition(),  
                "tipo declarado del campo " + campoDecl.getSimpleName()  
                + " (" + campoDecl.getType() + ")"  
                + " no implementa la interfaz " + this.tipoIContextoApp);  
        }  
    }  
}
```

10.5.4.- Validación avanzada: @JavaBean.

En el siguiente ejemplo vamos a profundizar en la funcionalidad de validación de código fuente del procesamiento de anotaciones.

Esta vez la validación no afectará a elementos de código terminales y aislados, si no que se realizará sobre el conjunto de toda la clase y requerirá de una mayor complejidad.

Requisitos:

Diseñar un tipo anotación aplicable exclusivamente sobre clases que permita comprobar si la clase anotada es un [Java Bean](#), es decir, que cumple con los siguientes requisitos:

- 1) La clase es serializable (es decir, que implementa la interfaz `java.io.Serializable`).
- 2) La clase tiene un constructor sin argumentos que permite crear instancias sin información.
- 3) Todos los campos de la clase son accesibles a través de métodos setter y getter. Contemplar que los métodos getter de los campos tipo `boolean` puedan empezar alternativamente por “is”.

En caso de que no se cumpla cualquiera de estos requisitos, el procesamiento de la anotación `@JavaBean` deberá emitir un error de compilación vinculado a la declaración de clase anotada.

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface JavaBean { }
```

Implementación:

Proyecto: `anotaciones-proc-java5`.

Paquete: `anotaciones.procesamiento.java5.ejemplos.javaBean`.

Clase anotada:

Se incluyen las clases anotadas `JavaBeanClaseAnotadaJavaBean` como ejemplo de clase Java Bean válida y `JavaBeanClaseAnotadaNoJavaBean` como ejemplo de clase Java que cumple algunos de los requisitos para ser Java Bean, pero no llega a serlo.

Invocación:

```
apt.exe -version ^
-XListAnnotationTypes -XListDeclarations -XPrintAptRounds -XPrintFactoryInfo ^
-cp ".\target\classes;.\lib\mirror-api-1.0.0.jar" ^
-s .\src\main\java-generated ^
-d .\target\classes ^
-factory anotaciones.procesamiento.java5.ejemplos.javaBean.JavaBeanProcessorFactory ^
.\src\main\java\anotaciones\procesamiento\java5\ejemplos\javaBean\*.java
```

Resultados:

Cuando se invoca el procesamiento de anotaciones sobre el código del paquete `javaBean`, los resultados son los esperados: se encuentran 2 declaraciones de clase anotadas. Una de ellas cumple los requisitos para ser Java Bean, y la otra no y el procesador arroja un error:

```
Note: procesador: anotaciones.procesamiento.java5.ejemplos.javaBean.JavaBeanProcessor
Note: procesador: iniciando procesador...
Note: procesador: declaraciones anotadas: recuperando declaraciones anotadas...
Note: procesador: declaraciones anotadas: 2 declaraciones anotadas.
Note: procesador: procesamiento: iniciando procesamiento...
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java5\src\main\java\anotaciones\procesamiento\java5\ejemplos\javaBean\JavaBeanClaseAnotad
aNoJavaBean.java:15: error: @JavaBean: error: la clase
anotaciones.procesamiento.java5.ejemplos.javaBean.JavaBeanClaseAnotadaNoJavaBean no tiene
setter y/o getter para los siguientes campos: [variableInstanciaInteger,
variableInstanciaBoolean]
public class JavaBeanClaseAnotadaNoJavaBean implements java.io.Serializable {
    ^
Note: procesador: procesamiento: procesamiento completado.
Note: procesador: anotaciones.procesamiento.java5.ejemplos.javaBean.JavaBeanProcessor
finalizado.
```

Comentarios:

En el código del `JavaBeanProcessor`, ya que el tipo anotación `@JavaBean` sólo se aplica a clases y nos da igual el orden, hemos utilizado el método `getDeclarationsAnnotatedWith` para traernos directamente todas las declaraciones anotadas en un orden indefinido.

Después, para cada declaración anotada, debemos comprobar que sea una clase y no una interfaz o un tipo enumerado. `@JavaBean` es `@Target(ElementType.TYPE)` (incluye clases, interfaces y enumerados) y los usuarios podrían haber colocado la anotación incorrectamente sobre interfaces o tipos enumerados, y el procesador deberá arrojar el correspondiente error:

```
// procesamos una a una y en un orden indeterminado (nos da igual el orden)
// todas las declaraciones anotadas con el tipo anotación a procesar
for (Declaration decl : declAnotadas) {

    // PROCESAMIENTO DE LA DECLARACIÓN

    TypeDeclaration tipoDecl = (TypeDeclaration) decl;

    // COMPROBACIÓN PREVIA: LA DECLARACIÓN NO ES UNA INTERFAZ O UN TIPO ENUMERADO

    // debido a que una TypeDeclaration puede corresponder a una interfaz
    // (incluso a un tipo anotación) o un tipo enumerado, hay que comprobarlo
    if ((tipoDecl instanceof InterfaceDeclaration) || (tipoDecl instanceof EnumDeclaration)) {

        // la anotación @JavaBean se ha puesto sobre una interfaz o un tipo enumerado:
        // arrojamos un error de compilación
        messenger.printError(tipoDecl.getPosition(),
            "@JavaBean: error: @JavaBean sólo es aplicable"
            + " a clases (" + tipoDecl + " no es una clase)");

        // y pasamos a procesar la siguiente declaración en la siguiente iteración del for
        continue;
    }

    // ... resto del código ...
}
```

Una vez hemos comprobado que la declaración es una declaración de clase (`claseDecl`), pasamos a comprobar cada uno de los requisitos exigidos para considerar la clase Java Bean:

```
// COMPROBACIÓN 1) SERIALIZABLE

// ¿implementa la clase la interfaz java.io.Serializable?
ClassType claseTipo = (ClassType) this.types.getDeclaredType(claseDecl);
if (!this.types.isAssignable(claseTipo, this.tipoInterfazSerializable)) {

    // la clase no implementa la interfaz java.io.Serializable -> ERROR
    messager.printError(
        claseDecl.getPosition(),
        "@JavaBean: error: la clase " + claseDecl
        + " no implementa java.io.Serializable");
}

// COMPROBACIÓN 2) CONSTRUCTOR SIN ARGUMENTOS

// recuperamos los constructores de la clase
Collection<ConstructorDeclaration> constructores = claseDecl.getConstructors();

// pasamos el filtro de constructores sin argumentos a los constructores de la clase
// y vemos si entre ellos no está el constructor sin argumentos que requerimos
if (this.filtroConstructorSinArgumentos.filter(constructores).size() == 0) {

    // entre la lista de constructores no está el constructor sin argumentos -> ERROR
    messager.printError(
        claseDecl.getPosition(),
        "@JavaBean: error: la clase " + claseDecl
        + " no tiene un constructor sin argumentos");
}

// COMPROBACIÓN 3) CAMPOS ACCESIBLES A TRAVÉS DE MÉTODOS SETTER Y GETTER

// obtenemos la lista de campos de la clase
Collection<FieldDeclaration> campos = claseDecl.getFields();

// comprobamos si hay campos que no tengan correctamente definidos su setter y getter
ArrayList<FieldDeclaration> camposSinSetterOGetter = new ArrayList<FieldDeclaration>();

for (FieldDeclaration campoDecl : campos) {

    // ¿carece el campo de método setter o de método getter?
    if (!campoTieneMetodoSetter(claseDecl, campoDecl)
        || !campoTieneMetodoGetter(claseDecl, campoDecl)) {

        // el campo carece de método getter o setter
        // -> lo añadimos a la lista de campos sin setter o getter
        camposSinSetterOGetter.add(campoDecl);
    }
}

// ¿hay campos que carezcan de métodos setter o getter?
if (!camposSinSetterOGetter.isEmpty()) {

    // si hay campos que carecen de métodos setter o getter -> ERROR
    messager.printError(
        claseDecl.getPosition(),
        "@JavaBean: error: la clase " + claseDecl
        + " no tiene setter y/o getter para "
        + "los siguientes campos: " + camposSinSetterOGetter);
}
```

Como podemos ver, el código anterior está estructurado en 3 bloques de no mucho tamaño. Esto se ha podido lograr a través del uso de filtros **DeclarationFilter**, que nos permiten seleccionar declaraciones sin tener que montar construcciones de bucles iterativos, así como de la refactorización de operaciones demasiado aparatosas a métodos auxiliares.

Especialmente interesantes son en este caso los métodos auxiliares **campoTieneMetodoSetter** y **campoTieneMetodoGetter**, ya que permiten simplificar la lógica de comprobación de si la clase define métodos getter y setter para cada uno de sus campos. En dichas operaciones también se han utilizado filtros para no tener que montar bucles iterativos. La estrategia implementada ha sido definir unos “filtros base” de métodos setter y getter, sobre los que luego hemos usado composición de filtros para crear filtros que tuvieran en cuenta las propiedades concretas de cada campo, como su nombre o su tipo.

A continuación se incluye el código del método auxiliar **campoTieneMetodoGetter**, que tiene como peculiaridad especial el hecho de tener que contemplar la posibilidad de que el getter de una propiedad booleana pueda empezar por “is”. Nótese la composición del filtro para el campo a partir del “filtro base” de métodos getters llamado **filtroMetodosGetter**. El método auxiliar correspondiente a los setter, **campoTieneMetodoSetter**, es muy similar en su construcción.

```

private boolean campoTieneMetodoGetter(
    ClassDeclaration claseDecl,
    final FieldDeclaration campoDecl) {

    // vrble resultado
    boolean campoTieneMetodoGetter = true;

    // nombres del método getter correspondiente al campo
    String nombreCampo = campoDecl.getSimpleName();
    String sufijoNombreGetter =
        nombreCampo.substring(0, 1).toUpperCase()
        + nombreCampo.substring(1);
    final String nombreGetter1 = "get" + sufijoNombreGetter;
    final String nombreGetter2 = "is" + sufijoNombreGetter;

    // filtro únicamente para el getter del campo
    // a partir del filtro de getters general
    DeclarationFilter filtroGetterCampo =
        this.filtroMetodosGetter.and(new DeclarationFilter() {
            @Override
            public boolean matches(Declaration decl) {
                MethodDeclaration methodDecl = (MethodDeclaration) decl;
                String nombreMetodo = methodDecl.getSimpleName();
                TypeMirror tipoRetorno = methodDecl.getReturnType();
                return (nombreMetodo.equals(nombreGetter1)
                    || nombreMetodo.equals(nombreGetter2))
                    && types.isSubtype(tipoRetorno, campoDecl.getType());
            }
        });
    });

    // para saber si el campo tiene un getter sólo tenemos
    // que aplicar el filtro a la lista de métodos de la clase
    campoTieneMetodoGetter =
        filtroGetterCampo.filter(claseDecl.getMethods()).size() > 0;

    // devolvemos el resultado
    return campoTieneMetodoGetter;
}

```

10.5.5.- Generación de código básica: @Proxy.

A continuación vamos a introducir la última de las tipologías de funcionalidad más importantes del procesamiento de anotaciones: la **generación de ficheros**.

Como ya hemos mencionado, la generación de ficheros, especialmente de ficheros de código fuente, fue una de las necesidades más importantes que vino a cubrir la introducción del procesamiento de anotaciones en el lenguaje Java y en las que más se pensó en un principio. Actualmente, no obstante, cada vez están más en boga las aplicaciones de los procesadores de anotaciones como validadores adicionales de semánticas complejas en tiempo de compilación

Para la generación de nuevos ficheros de código fuente durante el procesamiento de anotaciones, la Mirror API nos proporciona la interfaz **Filer** con métodos para crear ficheros Java de código o de clase, así como ficheros normales, tanto de texto como binarios.

Requisitos:

Diseñar un tipo anotación aplicable exclusivamente sobre clases que permita la creación de una subclase que añadirá estado adicional (en forma de una nueva variable de instancia), así como comportamiento adicional (añadiendo la implementación de una nueva interfaz).

Más concretamente, deben de cumplirse las siguientes condiciones:

- 1) El nombre de la nueva subclase será el nombre de la clase original más el sufijo “**Proxy**”.
- 2) El paquete de la nueva subclase será también el de la clase original más el sufijo “**Proxy**”.
- 3) La subclase añadirá una variable de tipo **EstadoProxy** e implementará la interfaz **IProxy**.

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface Proxy { }
```

Implementación:

Proyecto: **anotaciones-proc-java5**.

Paquete: **anotaciones.procesamiento.java5.ejemplos.proxy**.

Clase anotada:

Se incluye la clase anotada **ProxyClaseAnotada** que generará a **ProxyClaseAnotadaProxy** dentro del paquete **anotaciones.procesamiento.java5.ejemplos.proxyProxy**.

Invocación:

```
apt.exe -version ^
-XListAnnotationTypes -XListDeclarations -XPrintAptRounds -XPrintFactoryInfo ^
-cp ".\target\classes;.\lib\mirror-api-1.0.0.jar" ^
-s .\src\main\java-generated ^
-d .\target\classes ^
-factory anotaciones.procesamiento.java5.ejemplos.proxy.ProxyProcessorFactory ^
.\src\main\java\anotaciones\procesamiento\java5\ejemplos\proxy\*.java
```

Resultados:

Cuando se invoca el procesamiento de anotaciones sobre el código del paquete `proxy`, se encuentra una declaración anotada, la correspondiente a la clase `ProxyClaseAnotada`, y se genera el fichero correspondiente a su subclase, llamado `ProxyClaseAnotadaProxy` dentro del paquete `anotaciones.procesamiento.java5.ejemplos.proxyProxy`.

Lo más interesante del resultado es que, al haber generado nuevo código fuente, `apt` entra en una segunda ronda de procesamiento utilizando como entrada el nuevo fichero:

```
Note: procesador: anotaciones.procesamiento.java5.ejemplos.proxy.ProxyProcessor
Note: procesador: iniciando procesador...
Note: procesador: declaraciones anotadas: recuperando declaraciones anotadas...
Note: procesador: declaraciones anotadas: 1 declaraciones anotadas.
Note: procesador: procesamiento: iniciando procesamiento...
Note: procesador: procesamiento: clase:
anotaciones.procesamiento.java5.ejemplos.proxy.ProxyClaseAnotada
Note: procesador: procesamiento: fichero: generando fichero...
Note: procesador: procesamiento: fichero: abriendo fichero...
Note: procesador: procesamiento: fichero: fichero abierto.
Note: procesador: procesamiento: fichero: escribiendo fichero...
Note: procesador: procesamiento: fichero: fichero escrito.
Note: procesador: procesamiento: fichero: cerrando fichero...
Note: procesador: procesamiento: fichero: fichero cerrado.
Note: procesador: procesamiento: procesamiento completado.
Note: procesador: anotaciones.procesamiento.java5.ejemplos.proxy.ProxyProcessor
finalizado.
1 warning
apt Round : 2
filenames: [.\src\main\java-
generated\anotaciones\procesamiento\java5\ejemplos\proxyProxy\ProxyClaseAnotadaProxy.java
]
options: com.sun.tools.javac.util.Options@2ea37a18
Set of annotations found: []
Set of Specified Declarations:
[anotaciones.procesamiento.java5.ejemplos.proxyProxy.ProxyClaseAnotadaProxy]
Set of Included Declarations:
[anotaciones.procesamiento.java5.ejemplos.proxyProxy.ProxyClaseAnotadaProxy]
Factory anotaciones.procesamiento.java5.ejemplos.proxy.ProxyProcessorFactory matches
nothing.
```

Nótese que el nuevo fichero generado se ha guardado en el directorio `.\src\main\java-generated`, ya que en la invocación a `apt` especificamos la opción `-s .\src\main\java-generated`. En la invocación de los ejemplos anteriores, casi como si fuera una plantilla, habíamos especificado siempre el valor de la opción `-s`, pero, ya que en los ejemplos anteriores no se generaban nuevos ficheros, esa opción no servía a efectos prácticos para nada. En este ejemplo es donde tiene verdadero sentido dicha opción y se utiliza por primera vez.

En la segunda ronda, como vemos en la información de depuración mostrada, el conjunto de anotaciones encontradas es vacío (`Set of annotations found: []`), ya que la nueva clase no está anotada con ningún tipo anotación, con lo cual no hay nada que procesar y esta segunda ronda se convierte en la ronda final y `apt` termina.

Una vez `apt` ha terminado, como sabemos, se pasan todos los ficheros, los originales y los generados, al compilador de Java para ser compilados, guardándose todos los ficheros de clase compilados en la ruta dada por la opción `-d`, en este caso: `-d .\target\classes`.

Comentarios:

Al terminar el procesamiento de anotaciones del procesador **ProxyProcessor**, vemos que se ha generado un nuevo fichero **ProxyClaseAnotadaProxy.java** en el directorio de salida de ficheros generados, dado por la opción **-s .\src\main\java-generated**. Si exploramos el proyecto veremos que está en el directorio esperado.

Del mismo modo, dicho fichero **.java** ha sido compilado tras la última ronda de procesamiento y su fichero de clase **.class** compilado se encontrará en la ruta destino dada por la opción **-d .\target\classes**.

Si estuviéramos aún en fase de desarrollo del procesador de anotaciones, podríamos abrir el fichero **.java** generado en el entorno para así comprobar si se está generando el código fuente exacto que queremos y así corregir posibles errores. En este caso el fichero **.java** generado es el siguiente:

```
package anotaciones.procesamiento.java5.ejemplos.proxyProxy;

import anotaciones.procesamiento.java5.ejemplos.proxy.*;

/**
 * Clase proxy generada para la clase original anotaciones.procesamiento.java5.ejemplos.proxy.ProxyClaseAnotada.
 * Amplia la funcionalidad de la clase original con los métodos de la interfaz <code>IPrxy</code>.
 * Añade también la variable de estado pertinente para el funcionamiento del proxy <code>EstadoProxy</code>.
 * Fecha y hora de generación de este fichero: Mon May 26 01:06:39 CEST 2014
 */
public class ProxyClaseAnotadaProxy extends ProxyClaseAnotada implements IPrxy {

    // =====
    // VARIABLES DE INSTANCIA
    // =====

    // variable de estado para el proxy
    private EstadoProxy estadoProxy = new EstadoProxy();

    // =====
    // MÉTODOS DE APLICACIÓN
    // =====

    public void proxyMetodo1(String argumentoMetodo1) {

        // ... implementación del método 1 ...
        this.estadoProxy.setVrbloEstado1(argumentoMetodo1);
    }

    public void proxyMetodo2(Integer argumentoMetodo2) {

        // ... implementación del método 2 ...
        this.estadoProxy.setVrbloEstado2(argumentoMetodo2);
    }

    public Float proxyMetodo3(Long argumentoMetodo3) {

        // ... implementación del método 3 ...
        this.estadoProxy.setVrbloEstado3(
            (float) (2 * Math.PI * argumentoMetodo3));

        // devolvemos el resultado
        return this.estadoProxy.getVrbloEstado3();
    }
}
```



En cuanto al código del **ProxyProcessor**, ya que el tipo anotación **@Proxy** sólo se aplica a clases y nos da igual el orden, hemos vuelto a usar el método **getDeclarationsAnnotatedWith** para traernos directamente todas las declaraciones anotadas en un orden indefinido.

Para cada declaración anotada, comprobamos que sea una clase y no una interfaz o un tipo enumerado, ya que **@Proxy** es **@Target(ElementType.TYPE)** (incluye clases, interfaces y enumerados) y los usuarios podrían colocar la anotación incorrectamente sobre interfaces o tipos enumerados, caso en el cual el procesador deberá arrojar el correspondiente error.

A continuación, preparamos la escritura del código fuente creando una serie de variables que nos vendrán bien y que facilitarán la legibilidad del código del procesador. Estas variables tendrán las propiedades tanto de la clase original como de la clase nueva a generar:

```
// PREPARACIÓN DEL FICHERO DE LA NUEVA SUBCLASE

messager.printNotice("procesador: procesamiento: clase: " + claseDecl.getQualifiedName());

// propiedades de la clase original
String paqueteClaseOriginal = claseDecl.getPackage().getQualifiedName();
String nombreClaseOriginal = claseDecl.getSimpleName();
String nombreCanonicoClaseOriginal = paqueteClaseOriginal + "." + nombreClaseOriginal;

// propiedades de la nueva clase a generar
String paqueteNuevaClase = claseDecl.getPackage().getQualifiedName() + "Proxy";
String nombreNuevaClase = claseDecl.getSimpleName() + "Proxy";
String nombreCanonicoNuevaClase = paqueteNuevaClase + "." + nombreNuevaClase;
```

Utilizamos la interfaz **Filer** para crear el manejador **PrintWriter** **fichero** que nos permitirá escribir al flujo de salida del nuevo fichero de código fuente. La estructura de creación del fichero está rodeada de un bloque **try-catch** para capturar una posible **IOException** del fichero. Este es el esquema simplificado sin entrar en detalles de la escritura del código:

```
try {
    // CREACIÓN DEL FICHERO

    // creamos un manejador de fichero asociado al nombre canónico de la nueva clase
    messager.printNotice("procesador: procesamiento: fichero: abriendo fichero...");
    fichero = this.filer.createSourceFile(nombreCanonicoNuevaClase);
    messager.printNotice("procesador: procesamiento: fichero: fichero abierto.");

    // CÓDIGO FUENTE DE LA NUEVA SUBCLASE
    // ... escritura del código a fichero (ver página siguiente) ...

} catch (IOException e) {

    // ERROR DURANTE LA CREACIÓN DEL FICHERO

    // error durante el proceso de creación del fichero de la nueva clase -> ERROR
    this.messager.printError("@Proxy: error: error durante la "
        + "generación del fichero correspondiente a la "
        + "clase " + claseDecl.getQualifiedName() + " -> " + e);
} finally {

    // CIERRE DEL FICHERO

    // cerramos el flujo del fichero
    messager.printNotice("procesador: procesamiento: fichero: cerrando fichero...");
    if (fichero!=null) fichero.close();
    messager.printNotice("procesador: procesamiento: fichero: fichero cerrado.");
}
```

Dentro del bloque **try-catch**, tras la apertura del fichero, sólo resta escribir el código fuente de la clase que queremos generar al flujo de salida del fichero. Cuando terminemos, el fichero se cerrará con la llamada a `close` del bloque **finally**.

Todo el cuerpo de escritura del código de la nueva clase es un poco monótono y aparatoso, ya que consta de una llamada tras otra al método `fichero.println` para escribir el nuevo código:

```
// CÓDIGO FUENTE DE LA NUEVA SUBCLASE

messager.printNotice("procesador: procesamiento: fichero: escribiendo fichero...");

// paquete
fichero.println("package " + paqueteNuevaClase + ";" + "\n");

// imports
fichero.println("import " + paqueteClaseOriginal + ".*;" + "\n");

// cabecera javadoc de la nueva clase
fichero.println("/**");
fichero.println(" * Clase proxy generada para la clase original " + nombreCanonicoClaseOriginal + ".");
fichero.println(" * Amplia la funcionalidad de la clase original con los métodos de la interfaz
<code>IProxy</code>.");
fichero.println(" * Añade también la variable de estado pertinente para el funcionamiento del proxy
<code>EstadoProxy</code>.");
fichero.println(" * Fecha y hora de generación de este fichero: " + new Date());
fichero.println(" */");

// cabecera de la nueva clase
fichero.println("public class " + nombreNuevaClase + " extends " + nombreClaseOriginal + " implements IProxy
{");

// bloque de variables de instancia
fichero.println("    ");
fichero.println("    // =====");
fichero.println("    // VARIABLES DE INSTANCIA");
fichero.println("    // =====");
fichero.println("    ");

// variables de instancia: la variable de estado del proxy
fichero.println("        // variable de estado para el proxy");
fichero.println("        private EstadoProxy estadoProxy = new EstadoProxy();");

// bloque de métodos de aplicación
fichero.println("    ");
fichero.println("    // =====");
fichero.println("    // MÉTODOS DE APLICACIÓN");
fichero.println("    // =====");
fichero.println("    ");

// método: proxyMetodo1
fichero.println("        public void proxyMetodo1(String argumentoMetodo1) {" );
fichero.println("            ");
fichero.println("            // ... implementación del método 1 ...");
fichero.println("            this.estadoProxy.setVrbleEstado1(argumentoMetodo1);");
fichero.println("        }");
fichero.println("    ");

// método: proxyMetodo2
fichero.println("        public void proxyMetodo2(Integer argumentoMetodo2) {" );
fichero.println("            ");
fichero.println("            // ... implementación del método 2 ...");
fichero.println("            this.estadoProxy.setVrbleEstado2(argumentoMetodo2);");
fichero.println("        }");
fichero.println("    ");

// método: proxyMetodo3
fichero.println("        public Float proxyMetodo3(Long argumentoMetodo3) {" );
fichero.println("            ");
fichero.println("            // ... implementación del método 3 ...");
fichero.println("            this.estadoProxy.setVrbleEstado3("");
fichero.println("                (float) (2 * Math.PI * argumentoMetodo3));");
fichero.println("            ");
fichero.println("            // devolvemos el resultado");
fichero.println("            return this.estadoProxy.getVrbleEstado3();");
fichero.println("        }");
fichero.println("    ");
```

```
// cerramos el cuerpo de la clase  
fichero.println("}");  
  
messager.printNotice("procesador: procesamiento: fichero: fichero escrito.");
```

Como vemos, la generación del nuevo código fuente del fichero no es especialmente complicada. Sin embargo, suele ser habitual que deban escribirse códigos a veces de muy gran tamaño, monótonos, llenos de llamadas al método `println`, y muy farragosos. Además de generar una estructura muy pobre de código fuente, son difíciles de editar y mantener.

Para mejorar la estructura de este código podría refactorizarse la escritura del código fuente a un método auxiliar que se encargara de la escritura a fichero y tratar de sacar el texto del código fuente a escribir a una variable `String` o a un fichero de plantilla.

La utilización de plantillas en este tipo de procesadores que implementan una funcionalidad de generación de código fuente es algo muy interesante a contemplar. Si la generación de código es algo a pequeña escala, puede ser más sencillo escribir directamente a fichero o crear una pequeña solución de reemplazo de variables sobre cadenas de caracteres.

No obstante, sobre todo si necesitamos generar código fuente a gran escala, con varios tipos de ficheros a generar y/o de gran tamaño, existen multitud de [APIs de plantillas](#) que, a través de un motor de plantillas ([template engine](#)), permiten una cómoda generación de ficheros a través de la combinación de plantillas estáticas con datos o parámetros dinámicos. Algunas de las APIs de plantillas más conocidas y populares en la comunidad Java son [FreeMarker](#) o [Apache Velocity](#).

Más concretamente en nuestro ámbito, el procesamiento de anotaciones J2SE 1.5 y pensando justamente en facilitar la generación de código fuente sin inundar el código de los procesadores de anotaciones de llamadas a `println` y similares, se creó una API especialmente orientada al trabajo con plantillas conjuntamente con `apt` llamada [**apt-jelly**](#).

[**apt-jelly**](#) [<http://apt-jelly.sourceforge.net>] es un motor de plantillas que permite generar nuevos ficheros (código fuente, ficheros de configuración, etcétera) a partir de código fuente Java. [**apt-jelly**](#) soporta directamente el uso de plantillas de [FreeMarker](#) y [Apache Jelly](#), e indirectamente de plantillas de [Apache Velocity](#). No obstante, hay que decir que tanto el propio proyecto [**apt-jelly**](#) como Apache Jelly están actualmente en estado durmiente, sin actividad. [**apt-jelly**](#) no se actualiza desde 2008 y Apache Jelly desde 2010.

No vamos a profundizar más a este respecto, ya que el uso de motores de plantillas es algo más allá del ámbito de este manual, pero podría ser un ejercicio práctico interesante para el lector modificar el código del ejemplo para generar el mismo código utilizando una plantilla, ya sea usando [**apt-jelly**](#) o las facilidades oportunas de cualquier otra API más moderna.

10.5.6.- Generación de código avanzada: @AutoTest.

En el siguiente ejemplo vamos a profundizar en la funcionalidad del procesamiento de anotaciones que permite la generación de ficheros. En esta ocasión, la generación de código no será tan estática, si no que tendrá un componente más dinámico, pudiendo ocurrir que las clases generadas tengan más o menos métodos, así como los métodos cuerpos diferentes.

Requisitos:

Se quiere implementar una pequeña API de **generación automática de tests de unidad** que, a diferencia de otras como [JUnit](#), generará los tests automáticamente a partir de los casos de prueba indicados por un conjunto de anotaciones. Para ello, se deberá diseñar un tipo anotación **@AutoTest**, aplicable exclusivamente sobre métodos de clases, que permitirá definir casos de prueba sencillos para cada método mediante anotaciones **@AutoTestCasoPrueba**.

En el tipo anotación **@AutoTestCasoPrueba** se indicará la lista de argumentos del método, tal y como se escribirían en código Java, separados por la barra vertical “|”, así como el resultado concreto que se espera para dicho test. Se presentan adicionalmente los siguientes requisitos:

- 1) El nombre de la nueva subclase será el de la clase original más el sufijo “**AutoTest**”.
- 2) El paquete de la nueva subclase será el de la clase original más el sufijo “**AutoTest**”.
- 3) Los auto tests se generarán agrupados por clases en el orden dado por su nombre canónico.
- 4) Los auto tests deberán poder probar métodos de resultados variables, **void** o errores.
- 5) Deberá especificarse al menos un **@AutoTestCasoPrueba** o el procesador dará un error.
- 6) Los auto tests se anotarán con **@AutoTestGenerado**, un tipo anotación con retención **RUNTIME**, para poder ser descubiertos y ejecutados en tiempo de ejecución por un “test runner”.
- 7) El procesador informará por la salida del compilador de los auto tests según sean generados.

Esta pequeña API de generación automática de tests supone una combinación interesante de las capacidades de generación de ficheros de código con los mecanismos de reflexión y otras características de las anotaciones, como, por ejemplo, las anotaciones múltiples de **@AutoTest** como tipo anotación contenedor y **@AutoTestCasoPrueba** como anotación contenida.

No obstante, este ejemplo lógicamente adolece de muchas restricciones. La principal de ella es la imposibilidad de pasar argumentos de clases complejas a través de los argumentos de las propias anotaciones. Esto se debe a las propias restricciones que el lenguaje Java establece sobre los valores posibles que puede tener el elemento de una anotación.

[JUnit](#), por ejemplo, solventa el problema de la instanciación y población de valor de los objetos de clases no triviales a través de su tipo anotación **@Before**, que anota el método de la clase de test que se encargará de construir en su cuerpo los objetos complejos que los tests necesitan para trabajar. Así pues, los métodos etiquetados con **@Before** se llaman justo antes de cada test de la clase para dejar el estado del test donde se deseé. No obstante, como hemos dicho, esta API no genera los tests de forma automática, así que dicho código tendrá que ser escrito por los desarrolladores. En el caso de este ejemplo, a efectos meramente ilustrativos, hemos tratado de llegar todo lo lejos que fuera posible sin que el usuario de nuestros tipos anotación tuviera que escribir una sola línea de código.

Tipos anotación diseñados:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.METHOD)
public @interface AutoTest {

    AutoTestCasoPrueba[] casosPrueba(); // implicitamente se toma: default { };

}

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.METHOD)
public @interface AutoTestCasoPrueba {

    // =====
    // ELEMENTOS
    // =====

    String argumentos() default "";
    String resultado() default "";
    AutoTestTipoResultado tipoResultado() default AutoTestTipoResultado.VARIABLE;

    // =====
    // TIPOS ENUMERADOS ESTÁTICOS
    // =====

    public enum AutoTestTipoResultado { VARIABLE, VOID, ERROR }
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AutoTestGenerado { }
```

Implementación:

Proyecto: **anotaciones-proc-jav5**.

Paquete: **anotaciones.procesamiento.java5.ejemplos.autoTest**.

Clases anotadas:

Se han incluido las siguientes clases anotadas (dentro del paquete **autoTest**):

- **AutoTestClassAnotadaLecturaFicheros**
- **AutoTestClassAnotadaMultiplicarDividir**
- **AutoTestClassAnotadaSumarRestar**

Están listadas en el mismo orden lexicográfico ascendente en el que el procesador deberá procesarlas según los requisitos establecidos. Dichas clases generarán sus respectivas nuevas clases en **anotaciones.procesamiento.java5.ejemplos.autoTestAutoTest**.

Invocación:

Para este ejemplo tendremos 2 invocaciones: la invocación habitual de **apt** para la ejecución del procesamiento de anotaciones sobre el tipo anotación **@AutoTest**, y otra adicional que llamará al comando **java** (el entorno de ejecución de la Máquina Virtual Java) para ejecutar el código de la clase **AutoTestRunner**, una especie de código ejecutor de los auto tests generados durante el procesamiento, para que sean ejecutados y comprobar sus resultados directamente.

```
apt.exe -version ^
-XListAnnotationTypes -XListDeclarations -XPrintAptRounds -XPrintFactoryInfo ^
-cp ".\target\classes;.\lib\mirror-api-1.0.0.jar" ^
-s .\src\main\java-generated ^
-d .\target\classes ^
-factory anotaciones.procesamiento.java5.ejemplos.autoTest.AutoTestProcessorFactory ^
.\src\main\java\anotaciones\procesamiento\java5\ejemplos\autoTest\*.java

java.exe ^
-cp ".\target\classes;.\lib\mirror-api-1.0.0.jar" ^
anotaciones.procesamiento.java5.ejemplos.autoTest.AutoTestRunner
```

Resultados:

En cuanto al procesamiento de anotaciones, obtenemos la salida esperada, listándose las clases en orden lexicográfico ascendente, así como los casos de prueba que se van generando para ellas. Lo más interesante, al igual que en el ejemplo anterior del tipo anotación **@Proxy**, será comprobar como, en esta ocasión también se alcanza una segunda ronda de procesamiento con los tres ficheros nuevos generados.

A continuación sigue un resumen de la salida del procesador de anotaciones de **@AutoTest**:

```
Note: procesador: anotaciones.procesamiento.java5.ejemplos.autoTest.AutoTestProcessor
Note: procesador: iniciando procesador...
Note: procesador: declaraciones anotadas: recuperando declaraciones anotadas...
Note: procesador: declaraciones anotadas: 6 declaraciones anotadas.
Note: procesador: procesamiento: iniciando procesamiento...

... generación de las nuevas clases de métodos de prueba de las 2 primeras clases...

Note: procesador: procesamiento: clase: anotaciones.procesamiento.java5.ejemplos
.autoTest.AutoTestClaseAnotadaSumarRestar
Note: procesador: procesamiento: fichero: generando fichero...
Note: procesador: procesamiento: fichero: abriendo fichero...
Note: procesador: procesamiento: fichero: fichero abierto.
Note: procesador: procesamiento: fichero: escribiendo fichero...
Note: procesador: procesamiento: generando metodo de prueba: sumar_casoPrueba_1
Note: procesador: procesamiento: generando metodo de prueba: sumar_casoPrueba_2
Note: procesador: procesamiento: generando metodo de prueba: sumar_casoPrueba_3
Note: procesador: procesamiento: generando metodo de prueba: restar_casoPrueba_1
Note: procesador: procesamiento: generando metodo de prueba: restar_casoPrueba_2
Note: procesador: procesamiento: generando metodo de prueba: restar_casoPrueba_3
Note: procesador: procesamiento: fichero: fichero escrito.
Note: procesador: procesamiento: fichero: cerrando fichero...
Note: procesador: procesamiento: fichero: fichero cerrado.
Note: procesador: procesamiento: procesamiento completado.
Note: procesador: anotaciones.procesamiento.java5.ejemplos.autoTest.AutoTestProcessor
finalizado.
1 warning
```

```

apt Round : 2
filenames: [.\src\main\java-
generated\anotaciones\procesamiento\java5\ejemplos\autoTestAutoTest\AutoTestClaseAnotadaL
ecturaFicherosAutoTest.java, .\src\main\java-
generated\anotaciones\procesamiento\java5\ejemplos\autoTestAutoTest\AutoTestClaseAnotadaS
umarRestarAutoTest.java, .\src\main\java-
generated\anotaciones\procesamiento\java5\ejemplos\autoTestAutoTest\AutoTestClaseAnotadaM
ultiplicarDividirAutoTest.java]
options: com.sun.tools.javac.util.Options@794f47c
Set of annotations found:
[anotaciones.procesamiento.java5.ejemplos.autoTest.AutoTestGenerado]
Set of Specified Declarations:
[anotaciones.procesamiento.java5.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaLecturaFic
herosAutoTest,
anotaciones.procesamiento.java5.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaSumarRestar
AutoTest,
anotaciones.procesamiento.java5.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaMultiplicar
DividirAutoTest]
Set of Included Declarations:
[anotaciones.procesamiento.java5.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaLecturaFic
herosAutoTest,
anotaciones.procesamiento.java5.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaSumarRestar
AutoTest,
anotaciones.procesamiento.java5.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaMultiplicar
DividirAutoTest]
Factory anotaciones.procesamiento.java5.ejemplos.autoTest.AutoTestProcessorFactory
matches nothing.
warning: Annotation types without processors:
anotaciones.procesamiento.java5.ejemplos.autoTest.AutoTestGenerado
1 warning

```

Nótese efectivamente, como se ve en la información de depuración del principio de la página, cómo se entra en la segunda ronda de procesamiento. A diferencia de las clases proxy generadas en el ejemplo anterior, estas “clases AutoTest” sí están a su vez anotadas con el tipo anotación **@AutoTestGenerado**, pero este tipo anotación no es para ser procesado por algún otro procesador, si no que se utilizará como indicador para descubrir los métodos de auto tests en tiempo de ejecución con el **AutoTestRunner** y poder ejecutarlos y obtener sus resultados.

Ejecutando el comando que invoca al **AutoTestRunner** obtenemos la salida de los métodos de prueba y sus correspondientes resultados. Al final se muestra unos totales que informan del total de clases, tests ejecutados, test correctos OK y tests incorrectos NOOK (nótese que, aunque se ven errores durante la ejecución de los métodos de tests, el resultado de los tests es OK porque precisamente lo que se quiere probar en dichos tests es que los métodos originales fallan):

```

AutoTest runner: iniciando...
AutoTest runner: descubrimiento: paquete raiz: anotaciones.procesamiento
AutoTest runner: descubrimiento: clases AutoTest: 3
AutoTest runner: tests: iniciando ejecucion de tests...
AutoTest runner: tests: clase 1/3: anotaciones.procesamiento.java5.ejemplos.
autoTestAutoTest.AutoTestClaseAnotadaLecturaFicherosAutoTest
AutoTest: lecturaFicheroOK_casoPrueba_1: metodo void ejecutado sin errores.
AutoTest runner: tests: test lecturaFicheroOK_casoPrueba_1: OK
AutoTest: lecturaFicheroError_casoPrueba_1: error: java.io.FileNotFoundException: error:
lectura del fichero: C:\ruta\fichero\fichero_no_existente.txt
AutoTest runner: tests: test lecturaFicheroError_casoPrueba_1: OK

```

```

AutoTest runner: tests: clase 2/3: anotaciones.procesamiento.java5.ejemplos.
autoTestAutoTest.AutoTestClaseAnotadaMultiplicarDividirAutoTest
AutoTest: multiplicar_casoPrueba_1: resultado = 1
AutoTest runner: tests: test multiplicar_casoPrueba_1: OK
AutoTest: multiplicar_casoPrueba_2: resultado = 2
AutoTest runner: tests: test multiplicar_casoPrueba_2: OK
AutoTest: multiplicar_casoPrueba_3: resultado = 3
AutoTest runner: tests: test multiplicar_casoPrueba_3: OK
AutoTest: dividir_casoPrueba_1: resultado = 1
AutoTest runner: tests: test dividir_casoPrueba_1: OK
AutoTest: dividir_casoPrueba_2: resultado = 2
AutoTest runner: tests: test dividir_casoPrueba_2: OK
AutoTest: dividir_casoPrueba_3: resultado = 3
AutoTest runner: tests: test dividir_casoPrueba_3: OK
AutoTest: dividir_casoPrueba_4: error: java.lang.ArithmetricException: / by zero
AutoTest runner: tests: test dividir_casoPrueba_4: OK
AutoTest runner: tests: clase 3/3: anotaciones.procesamiento.java5.ejemplos.
autoTestAutoTest.AutoTestClaseAnotadaSumarRestarAutoTest
AutoTest: sumar_casoPrueba_1: resultado = 1
AutoTest runner: tests: test sumar_casoPrueba_1: OK
AutoTest: sumar_casoPrueba_2: resultado = 2
AutoTest runner: tests: test sumar_casoPrueba_2: OK
AutoTest: sumar_casoPrueba_3: resultado = 3
AutoTest runner: tests: test sumar_casoPrueba_3: OK
AutoTest: restar_casoPrueba_1: resultado = 0
AutoTest runner: tests: test restar_casoPrueba_1: OK
AutoTest: restar_casoPrueba_2: resultado = 1
AutoTest runner: tests: test restar_casoPrueba_2: OK
AutoTest: restar_casoPrueba_3: resultado = 2
AutoTest runner: tests: test restar_casoPrueba_3: OK
AutoTest runner: tests: ejecucion de tests finalizada.
AutoTest runner: tests: resultados: CLASES: 3
AutoTest runner: tests: resultados: TESTS: 15
AutoTest runner: tests: resultados: OK: 15
AutoTest runner: tests: resultados: NOOK: 0
AutoTest runner: finalizado.

```

Podemos hacer que falle cualquiera de estos tests si alteramos los casos de prueba a valores absurdos. Por ejemplo, si en la clase AutoTestClaseAnotadaSumarRestar cambiamos el resultado de cualquier caso de prueba a 666, el método de test correspondiente fallará si realmente el caso de prueba no se cumple:

```

@AutoTest(casosPrueba={
    @AutoTestCasoPrueba(argumentos="0|1", resultado="1", tipoResultado=AutoTestTipoResultado.VARIABLE),
    @AutoTestCasoPrueba(argumentos="1|1", resultado="2", tipoResultado=AutoTestTipoResultado.VARIABLE),
    @AutoTestCasoPrueba(argumentos="1|2", resultado="666", tipoResultado=AutoTestTipoResultado.VARIABLE)
})
public int sumar(int arg1, int arg2) {
    return arg1 + arg2;
}

```

El tercer caso de prueba del método sumar fallará porque $1+2$ no suma 666. Si lanzamos todo el proceso de nuevo, obtenemos el siguiente resultado para el tercer caso de prueba de sumar:

```

AutoTest: sumar_casoPrueba_3: resultado = 3
AutoTest runner: tests: test sumar_casoPrueba_3: NOOK

```

El tipo de resultado de los tests, tal y como se definió en sus requisitos, es de tres tipos posibles: **VARIABLE**, **VOID** y **ERROR**. Para ilustrar su funcionamiento, se ha definido la clase anotada **AutoTestClaseAnotadaLecturaFicheros**, con un método que no devuelve nada cuando se ejecuta (tipo de retorno **void**) y otro que está previsto que dé error al ejecutarse, ya que en el test se le pasará la ruta a un fichero inexistente.

```
public class AutoTestClaseAnotadaLecturaFicheros {  
  
    // ======  
    // MÉTODOS DE ACCESO  
    // ======  
  
    @AutoTest(casosPrueba={  
        @AutoTestCasoPrueba(argumentos = "\"C:\\\\ruta\\\\fichero\\\\fichero_existente.txt\"",  
                            resultado = "se ignora cuando el tipo de resultado es VOID",  
                            tipoResultado = AutoTestTipoResultado.VOID),  
    })  
    public void lecturaFicheroOK(String rutaFichero) throws IOException {  
  
        // ... implementación de la lectura del fichero ...  
    }  
  
    @AutoTest(casosPrueba={  
  
        @AutoTestCasoPrueba(argumentos = "\"C:\\\\ruta\\\\fichero\\\\fichero_no_existente.txt\"",  
                            resultado = "java.io.FileNotFoundException",  
                            tipoResultado = AutoTestTipoResultado.ERROR),  
    })  
    public void lecturaFicheroError(String rutaFichero) throws FileNotFoundException {  
  
        // ... implementación de la lectura del fichero ...  
        throw new FileNotFoundException("error: lectura del fichero: " + rutaFichero);  
    }  
}
```

En el tipo de resultado **VARIABLE**, como ya hemos visto, el valor del elemento resultado de la anotación **@AutoTestCasoPrueba** es el valor esperado de retorno para la ejecución del método a probar. En **VOID**, el valor del elemento resultado se ignora, ya que no tiene utilidad. Y en **ERROR** en el elemento resultado se deberá colocar el nombre canónico de la excepción que se espera que se arroje al ejecutar el método a probar.

Por ejemplo, en el caso del caso de prueba del método **lecturaFicheroError**, un test con tipo de resultado **ERROR**, en resultado se establece “**java.io.FileNotFoundException**”, que es el nombre canónico de la excepción que se espera que el método arroje al tratar de leer un fichero en una ruta que no existe (este test sirve por tanto para probar que el método exhibe un comportamiento deseable ante un escenario de error concreto).

Finalmente, comentar que el cuerpo del procesador AutoTestProcessor es bastante similar al del ejemplo de generación de código del tipo anotación `@Proxy`, salvo por el hecho único de que se ha hecho una “primera pasada” por las declaraciones anotadas para poder extraer de ellas la lista de clases anotadas con al menos una anotación `@AutoTest` válida, es decir: que no estuviera asociada a una interfaz o un tipo enumerado.

Nótese cómo se ha utilizado específicamente una colección de tipo TreeMap (ordenada) con el nombre canónico de las clases como clave. Esto permitirá posteriormente recorrer la lista de clases anotadas en orden lexicográfico ascendente según su nombre canónico tal y como piden los requisitos.

```
// PASADA INICIAL DE BÚSQUEDA DE CLASES CON MÉTODOS ANOTADOS

// realizamos una pasada por las declaraciones anotadas para guardar una
// lista ORDENADA de las clases que contienen al menos un método anotado
TreeMap<String, ClassDeclaration> declsClasesConMetodosAnotados = null;
declsClasesConMetodosAnotados = new TreeMap<String, ClassDeclaration>();

for (Declaration decl : declAnotadas) {

    // PROCESAMIENTO DE DECLARACIONES SUELTADES DESORDENADAS

    // sabemos que la declaración es una declaración de método -> down-cast
    MethodDeclaration metodoDecl = (MethodDeclaration) decl;

    // COMPROBACIÓN: TIPO DECLARANTE NO ES UNA INTERFAZ O UN ENUMERADO

    // obtenemos el tipo que declara este método (recordemos que tiene
    // que ser exclusivamente una clase; nada de interfaces o enumerados)
    TypeDeclaration tipoDecl = metodoDecl.getDeclaringType();

    // ¿el tipo donde de está el método anotado es una interfaz o un enumerad?
    if (    (tipoDecl instanceof InterfaceDeclaration)
        || (tipoDecl instanceof EnumDeclaration)) {

        // la anotación @AutoTest se ha puesto sobre un método declarado
        // dentro de una interfaz o un tipo enumerado -> ERROR
        messenger.printError(
            tipoDecl.getPosition(),
            "@AutoTest: error: @AutoTest sólo es aplicable"
            + " a métodos de clases (" + tipoDecl + " no es una clase)");

        // y pasamos a procesar la siguiente declaración
        continue;
    }

    // efectivamente la anotación está hecha sobre una clase -> down-cast
    ClassDeclaration claseDecl = (ClassDeclaration) tipoDecl;

    // DECLARACIÓN OK -> GUARDAMOS LA CLASE DONDE SE DECLARA LA ANOTACIÓN

    // añadimos dicha clase al map ordenado de clases con métodos anotados
    declsClasesConMetodosAnotados.put(claseDecl.getQualifiedName(), claseDecl);
}
```

La otra gran particularidad del cuerpo de la implementación de AutoTestProcessor es que, debido a la mayor complejidad de la generación de código necesaria, se ha refactorizado la generación del código fuente a un método auxiliar llamado generarMetodoPrueba que genera 3 tipos de métodos de prueba con ligeras diferencias en función de que el tipo de resultado sea **VARIABLE**, **VOID** o **ERROR**.

Esto es así porque en el caso de que el resultado sea **VARIABLE** habrá que recogerlo y compararlo con el resultado obtenido de la invocación del método original con sus correspondientes argumentos. En caso de ser **VOID**, no tiene sentido recoger ningún resultado y simplemente se retorna true si no ha habido ninguna excepción y, finalmente, en caso de ser un resultado de tipo **ERROR**, hay que comparar el nombre canónico de la clase excepción capturada con el valor dado por el elemento resultado.

```
private void generarMetodoPrueba(
    MethodDeclaration metodoDecl,
    AutoTestCasoPrueba casoPrueba,
    String nombreMetodoPrueba,
    PrintWriter fichero) {

    // propiedades útiles para la generación del cuerpo del método
    String nombreMetodo = metodoDecl.getSimpleName();
    String nombreClaseObjetoAProbar = metodoDecl.getDeclaringType().getSimpleName();
    String nombreClaseResultado = metodoDecl.getReturnType().toString();
    String[] argumentosArray = casoPrueba.argumentos().split("\\\\|");
    String argumentos = Arrays.toString(argumentosArray).replace("[", "").replace("]", "");
    String resultado = casoPrueba.resultado();

    // GENERACIÓN DEL CUERPO DEL MÉTODO EN FUNCIÓN DEL TIPO DE RESULTADO ESPERADO

    if (casoPrueba.tipoResultado() == AutoTestTipoResultado.VARIABLE) {

        // TIPO DE RESULTADO: VARIABLE

        // ... tipo de cuerpo para un método con un resultado de valor variable ...

    } else if (casoPrueba.tipoResultado() == AutoTestTipoResultado.VOID) {

        // TIPO DE RESULTADO: VOID

        // ... tipo de cuerpo para un método con un resultado sin valor de retorno ...

    } else if (casoPrueba.tipoResultado() == AutoTestTipoResultado.ERROR) {

        // TIPO DE RESULTADO: ERROR

        // ... tipo de cuerpo para un método con un resultado esperado de error ...

    }
}
```

10.6.- Migración de procesamiento J2SE 1.5 a JSR 269.

Como sabemos, el procesamiento de anotaciones como se hacía en J2SE 1.5, usando `apt` y la Mirror API, fue deprecada ya en Java SE 6, con la aparición de la especificación JSR 269.

En el [blog de Joseph Darcy](#), ingeniero jefe creador de `apt`, en su entrada del 27 de Julio del año 2009 titulada “[An apt replacement](#)”, anunciaba la eliminación definitiva de `apt` tras su última aparición en la JDK de Java SE 7. Al mismo tiempo, y con motivo de la noticia, publicaba unas tablas muy útiles para los desarrolladores que quieran portar sus procesadores de anotaciones de la Mirror API a la API de procesamiento JSR 269. Dada su especial relevancia, reproducimos dichas tablas a continuación (hay una tabla por cada paquete de la Mirror API):

Mirror API - com.sun.mirror.apt	JSR 269 - Equivalente aproximado
<code>AnnotationProcessor</code>	<code>javax.annotation.processing.Processor</code>
<code>AnnotationProcessorEnvironment</code>	<code>javax.annotation.processing.ProcessingEnvironment</code>
<code>AnnotationProcessorFactory</code>	<code>javax.annotation.processing.Processor</code>
<code>AnnotationProcessorListener</code>	No existe un equivalente.
<code>AnnotationProcessors</code>	No existe un equivalente.
<code>Filer</code>	<code>javax.annotation.processing.Filer</code>
<code>Filer.Location</code>	<code>javax.tools.StandardLocation</code>
<code>Messager</code>	<code>javax.annotation.processing.Messager</code>
<code>RoundCompleteEvent</code>	No existe un equivalente.
<code>RoundCompleteListener</code>	No existe un equivalente.
<code>RoundState</code>	<code>javax.annotation.processing.RoundEnvironment</code>

Mirror API - com.sun.mirror.declaration	JSR 269 - Equivalente aproximado
<code>AnnotationMirror</code>	<code>javax.lang.model.element.AnnotationMirror</code>
<code>AnnotationTypeDeclaration</code>	<code>javax.lang.model.element.TypeElement</code>
<code>AnnotationTypeElementDeclaration</code>	<code>javax.lang.model.element.ExecutableElement</code>
<code>AnnotationValue</code>	<code>javax.lang.model.element.AnnotationValue</code>
<code>ClassDeclaration</code>	<code>javax.lang.model.element.TypeElement</code>
<code>ConstructorDeclaration</code>	<code>javax.lang.model.element.ExecutableElement</code>
<code>Declaration</code>	<code>javax.lang.model.element.Element</code>
<code>EnumConstantDeclaration</code>	<code>javax.lang.model.element.VariableElement</code>
<code>EnumDeclaration</code>	<code>javax.lang.model.element.TypeElement</code>
<code>ExecutableDeclaration</code>	<code>javax.lang.model.element.ExecutableElement</code>
<code>FieldDeclaration</code>	<code>javax.lang.model.element.VariableElement</code>
<code>InterfaceDeclaration</code>	<code>javax.lang.model.element.TypeElement</code>
<code>MemberDeclaration</code>	<code>javax.lang.model.element.Element</code>
<code>MethodDeclaration</code>	<code>javax.lang.model.element.ExecutableElement</code>
<code>Modifier</code>	<code>javax.lang.model.element.Modifier</code>
<code>PackageDeclaration</code>	<code>javax.lang.model.element.PackageElement</code>
<code>ParameterDeclaration</code>	<code>javax.lang.model.element.VariableElement</code>
<code>TypeDeclaration</code>	<code>javax.lang.model.element.TypeElement</code>
<code>TypeParameterDeclaration</code>	<code>javax.lang.model.element.TypeParameterElement</code>

Mirror API - com.sun.mirror.type	JSR 269 - Equivalente aproximado
AnnotationType	javax.lang.model.type.DeclaredType
ArrayType	javax.lang.model.type.ArrayType
ClassType	javax.lang.model.type.DeclaredType
DeclaredType	javax.lang.model.type.DeclaredType
EnumType	javax.lang.model.type.DeclaredType
InterfaceType	javax.lang.model.type.DeclaredType
MirroredTypeException	javax.lang.model.type.MirroredTypeException
MirroredTypesException	javax.lang.model.type.MirroredTypesException
PrimitiveType	javax.lang.model.type.PrimitiveType
PrimitiveType.Kind	javax.lang.model.type.TypeKind
ReferenceType	javax.lang.model.type.ReferenceType
TypeMirror	javax.lang.model.type.TypeMirror
TypeVariable	javax.lang.model.type.TypeVariable
VoidType	javax.lang.model.type.NoType
WildcardType	javax.lang.model.type.WildcardType

Mirror API - com.sun.mirror.util	JSR 269 - Equivalente aproximado
DeclarationFilter	javax.lang.model.util.ElementFilter
DeclarationScanner	javax.lang.model.util.ElementScanner6
DeclarationVisitor	javax.lang.model.element.ElementVisitor
DeclarationVisitors	No existe un equivalente.
Declarations	javax.lang.model.util.Elements
SimpleDeclarationVisitor	javax.lang.model.util.SimpleElementVisitor6
SimpleTypeVisitor	javax.lang.model.util.SimpleTypeVisitor6
SourceOrderDeclScanner	javax.lang.model.util.SimpleElementVisitor6
SourcePosition	No existe un equivalente.
TypeVisitor	javax.lang.model.element.TypeVisitor
Types	javax.lang.model.util.Types

Como se ve en las tablas, la Mirror API original se ha escindido en dos: las entidades que modelaban el lenguaje Java por un lado y las modelaban el comportamiento de `apt` por otro.

Mirror API	JSR 269
com.sun.mirror.(declaration/type/util)	javax.lang.model.(element/type/util)
com.sun.mirror.apt	javax.annotation.processing

De esta forma, la Mirror API se ha convertido en dos APIs distintas, desacoplando con buen criterio lo que eran dos cosas diferentes. Así, ahora `javax.lang.model`, aunque siga vinculada al procesamiento de anotaciones, se concibe como una API de propósito más general que modela el lenguaje Java. Y para el procesamiento está `javax.annotation.processing`.

Además, por el camino se han quedado las factorías de procesadores, cuyas funciones se han integrado en los propios procesadores JSR 269 y los listeners, innecesarios al estar ahora su información siempre disponible. También desaparece `SourcePosition`, ya que las posiciones de línea/columna eran muy costosas de calcular y no eran una buena representación canónica.

10.7.- aptIn16 (apt en Java SE 6+).

aptIn16 [<https://github.com/moparisthebest/aptIn16>] es una implementación de **apt** y las clases de la Mirror API que permite ejecutar los procesadores de anotaciones J2SE 1.5 en Java SE 6 dentro del compilador **javac** como un procesador JSR 269 llamado **AptProcessor**.

aptIn16 implementa lo que podría llamarse un puente que permite interoperar entre la Mirror API y las APIs JSR 269. Es decir, que **aptIn16** implementa la funcionalidad original que proporcionaban **apt** y la Mirror API utilizando para ello las APIs de procesamiento JSR 269 (**javax.lang.model** y **javax.annotation.processing**). Este es un ejemplo del patrón de diseño “Adapter”, donde la API cliente (Client) sería la Mirror API, el adaptador (Adapter) sería **aptIn16** y el adaptado (Adaptee) las APIs de procesamiento JSR 269. La verdad es que hacer dicha adaptación es un truco muy interesante si se quiere a toda costa no perder el trabajo realizado con **apt** teniendo que portarlo todo a las nuevas APIs de procesamiento JSR 269.

IMPORTANTE: **aptIn16** se incluye en este manual por completitud y por su marcado interés didáctico, pero no debería contemplarse como una solución segura a largo plazo para la migración del procesamiento J2SE 1.5, si no, en todo caso, como una solución temporal de transición. No todas las facilidades originales están soportadas y algunos procesadores J2SE 1.5 podrían fallar. Además, el proyecto no se encuentra ya bajo un nivel de desarrollo activo.

En el código de **aptIn16** se incluyen explícitamente como dependencias todas las clases originales de la Mirror API. Luego, para poder interoperar con las facilidades de procesamiento JSR 269 se han dispuesto una serie de clases **ConvertX** que convierten la funcionalidad de las clases antiguas de la Mirror API a la funcionalidad de las clases correspondientes de las APIs JSR 269. Por poner un ejemplo **ConvertProcessingEnvironment** convierte o adapta la vieja interfaz **AnnotationProcessorEnvironment** de la Mirror API a la nueva **ProcessingEnvironment** de JSR 269. Esto ha sido posible, por supuesto, porque las nuevas APIs de procesamiento JSR 269 están fuertemente inspiradas y basadas en la Mirror API original, como hemos podido comprobar en el apartado anterior “Migración de procesamiento J2SE 1.5 a JSR 269”.

aptIn16 - Componentes principales	
Componente	Descripción
apt-mirror-api	Clases originales de la Mirror API. Son una dependencia necesaria para la compilación.
apt-processor	Contiene la clase com.moparisthebest.mirror.AptProcessor , que implementa un procesador moderno JSR 269 que ejecuta el mismo comportamiento que realizaría apt .
core	Contiene una clase ConvertX prácticamente para cada clase original de la Mirror API. Estas clases implementan la funcionalidad de las clases originales de la Mirror API, pero utilizando internamente las facilidades de las nuevas APIs de procesamiento JSR 269.

Así pues, al final lo que tenemos es un procesador JSR 269 llamado **AptProcessor** que, adheriéndose a las facilidades de las nuevas APIs JSR 269 y corriendo en **javac** como cualquier otro procesador JSR 269, ejecuta los mismos procesos que **apt**: descubrimiento de factorías, de anotaciones, emparejamiento, etcétera, de forma que el procesamiento se lleve a cabo de la forma más transparente posible para los procesadores originales J2SE 1.5.

Finalmente comentar que, por lo visto, **aptIn16** es mucho más rápido que **apt**, ya que funciona sobre el nuevo y optimizado **javac**. En su página web oficial, se comenta que una compilación de 13 minutos con el **apt** original se redujo a 45 segundos con **aptIn16**.

11.- NOVEDADES EN JAVA SE 6.

11.1.- Novedades generales en Java SE 6.

El 11 de Diciembre de 2006, Sun Microsystems liberaba públicamente **Java SE 6** (con el nombre o *codename* de “**Mustang**”). Desarrollada bajo la “especificación paraguas” [JSR 270](#), Sun decidió eliminar la numeración **1.x.0** del número de versión oficial (aunque internamente el número de versión seguía siendo **1.6.0**). También se decidió eliminar el 2, que se arrastraba desde J2SE 1.2 en el año 1998, y dejar las respectivas ediciones de la plataforma de desarrollo como Java SE (Standard Edition), Java EE (Enterprise Edition) y Java ME (Micro Edition).

Java SE 6 sería la última versión que publicaría Sun Microsystems, ya que, en Enero de 2010, fue adquirida por Oracle, pasando a esta última compañía toda la tecnología relacionada con Java. No obstante, antes de ser adquirida por Oracle, en Mayo de 2007, Sun había liberado como *open source* la mayor parte del código fuente de Java SE y creó el portal [OpenJDK](#) para conducir el desarrollo de parte de Java SE 6 con la comunidad de código abierto. Java SE 7 sería desarrollada ya completamente a través de OpenJDK, aunque bajo la estrecha tutela de Oracle.

En Java SE 6 no hubo ningún añadido al lenguaje Java. Se concentraron los esfuerzos en la plataforma de desarrollo Java, siguiendo varias líneas de trabajo muy definidas:

- Rendimiento: Nuevas estrategias y algoritmos de optimización.
- Interfaces gráficas: Swing: SwingWorker, operaciones de tablas, doble buffer.
- Utilidades: JDBC 4.0 para bases de datos y JAX-WS para servicios web.
- Herramientas (“tooling”): Java Compiler API (JSR 199) y Compiler Tree API.
- Anotaciones: Anotaciones estándar con las Commons Annotations (JSR 250).
- Anotaciones: Procesamiento estándar con Pluggable Annotation Processing API (JSR 269).

Las tres últimas son de especial interés para nosotros: las nuevas APIs de integración con herramientas de desarrollo (“tooling”) y, por supuesto, las anotaciones estándar Commons Annotations (JSR 250) y la API estándar de procesamiento de anotaciones JSR 269. En los siguientes apartados iremos profundizando en cada una de ellas.

Ofrecer mayor acceso a las facilidades de las herramientas de desarrollo internas de la plataforma Java posibilita desarrollos mejores, más sencillos, eficientes y estandarizados de herramientas de desarrollo (“tooling”) para Java. Además, ofrece una mayor integración de los servicios internos de la plataforma Java con aplicaciones relacionadas con el desarrollo, como los entornos de desarrollo integrado (Integrated Development Environments, o IDEs).

Este propósito de mejorar la integración de las herramientas y aplicaciones con la propia plataforma de desarrollo Java se llevó a cabo mediante una estandarización y publicación de ciertas facilidades hasta entonces de uso interno y, por tanto, no estándar, en APIs públicas.

No obstante, aunque todas las nuevas APIs de Java SE 6 eran públicas y pasaban a estar documentadas, algunas serían estándar y otras no. En cuanto a APIs estándar son prácticamente todas las nuevas. Especialmente importante es el caso de la **Java Compiler API**, al referirse a una pieza central de la plataforma de desarrollo Java como es el compilador, y que pasó por un proceso de estandarización a través del JCP (Java Community Process), para convertirse en la especificación [JSR 199](#). Por otro lado, en lo que respecta a APIs no estándar, tenemos el caso de la **Compiler Tree API**, que sólo está soportada por el compilador `javac` de Sun/Oracle.

11.2.- Common Annotations (JSR 250).

Para facilitar el desarrollo y promover la reutilización de código dentro de las diferentes plataformas Java SE y Java EE, y así evitar duplicidades, se introdujo la especificación **JSR 250** (**Common Annotations** for the Java™ Platform).

Las Commons Annotations, tal y como indica su nombre, son “anotaciones comunes” que responden a necesidades habituales de los procesos de desarrollo y que se estandarizan para que cada desarrollador no tenga que “reinventar la rueda” por su parte, si no que pueda utilizar una de las anotaciones estándar proporcionadas por la plataforma Java. Lógicamente, siempre es preferible adoptar una tecnología estándar, que esté bien respaldada y soportada por una amplia comunidad de desarrolladores, que desarrollar un tipo anotación propio, con la consiguiente carga de trabajo y mantenimiento que ello implica.

A las Common Annotations iniciales, aparecidas para Java SE 6 y Java EE 5, se irían añadiendo progresivamente en futuras versiones de Java SE y Java EE nuevos tipos anotación en los paquetes **javax.annotation**, **javax.annotation.security** y **javax.annotation.sql**, a medida que se fueron identificando necesidades de estandarización de tareas o funcionalidades comunes requeridos por la comunidad. Para más información sobre las Commons Annotations, puede consultarse el siguiente apartado del presente manual dedicado a las mismas.

11.3.- Java Compiler API (JSR 199).

La Java Compiler API (JSR 199) ofrece un completo acceso programático a todas las capacidades del compilador **javac**. La API anterior que podía ser utilizada para invocar el compilador (**com.sun.tools**) no era ni estándar, ni pública y, por tanto, estaba sujeta a posibles cambios, además de no estar correctamente documentada.

La principal funcionalidad que ofrece la Java Compiler API es que haciendo uso de ella es posible construir una invocación o tarea de compilación para realizar una **compilación de código totalmente personalizada desde dentro de una aplicación Java**, por lo que la compilación se convierte en un servicio que es posible realizar a nivel de aplicación.

11.4.- Compiler Tree API.

La Compiler Tree API ofrece facilidades de navegación por el código fuente visto desde un punto de vista sintáctico mediante el uso de estructuras jerárquicas llamadas **árboles sintácticos abstractos** (conocidos en inglés como AST = Abstract Syntax Trees). Estas facilidades, por supuesto, están encaminadas a facilitar enormemente la creación de APIs y herramientas de análisis de código, así como entornos de desarrollo (IDEs).

La Compiler Tree API no es estándar: sólo está oficialmente soportada por el compilador javac de Sun/Oracle. No es una API estándar oficial de la plataforma Java, ya que no ha pasado a través del JCP, por lo que su paquete raíz no empieza por **java** o **javax**, como es habitual para las APIs estándar, sino que hace referencia a Sun (**com.sun.source**).

IMPORTANTE: La Compiler Tree API es una API muy importante para el procesamiento de anotaciones ya que, como veremos más adelante, gracias a sus facilidades seremos capaces de descubrir anotaciones sobre variables locales.

11.5.- Pluggable Annotation Processing API (JSR 269).

En base a todos los problemas detectados en el soporte del procesamiento de anotaciones de J2SE 1.5, la comunidad Java notó que el procesamiento se hacía incómodo, engorroso, y tenía una gran cantidad de errores y limitaciones.

Se concluyó pues que era necesario un rediseño de la arquitectura del procesamiento de anotaciones, ya que incluso **la mera existencia de apt estaba fuera de lugar**, puesto que era más lógico que el procesamiento de anotaciones se hiciera dentro del propio compilador. Llegado este punto se identificó pues la necesidad de diseñar algún mecanismo que permitiera **integrar el procesamiento de anotaciones directamente en el compilador Java**. Además, **apt** y la Mirror API eran herramientas y APIs no estándar ofrecidas por Sun y que no formaban parte de la plataforma de desarrollo Java, por lo que convenía crear un estándar para facilitar el desarrollo de procesadores de anotaciones en Java.

Este propósito de estandarizar e integrar el procesamiento de anotaciones en el compilador se materializó en la JSR 269 (Pluggable Annotation Processing API), que entró a formar parte del proceso del JCP en Febrero de 2005 y su revisión final quedó definitivamente aprobada en Diciembre de 2006, con motivo del lanzamiento de **Java SE 6**.

Con Java SE 6, por tanto, el compilador **javac** fue rediseñado siguiendo los cánones de la especificación **JSR 269** para integrar el procesamiento de anotaciones como una parte más del proceso de compilación habitual, con lo cual, **a partir de Java SE 6, la herramienta apt ya no era necesaria** y, de hecho, aunque en Java SE 7 aún seguía presente, en Java SE 8 fur finalmente eliminada de la distribución de la plataforma.

Las principales **novedades introducidas por la especificación JSR 269** son:

- Compilador: Procesamiento de anotaciones integrado en el proceso de compilación.
- Compilador: Nuevas opciones: **proc**, **processor**, **processorpath** y opciones **Aclave [=valor]**
- API: Nueva API **javax.annotation.processing**: Contiene clases para definir procesadores de anotaciones y establecer comunicación entre estos y el entorno de procesamiento.
- API: Nueva API **javax.lang.model**: Contiene clases que modelan el lenguaje Java para así poder referenciar los diferentes tipos de elementos de código Java sobre los que se implementa la lógica de procesamiento de anotaciones.

Para más detalles sobre el procesamiento de anotaciones JSR 269, el presente manual le dedica más adelante un apartado monográfico “Procesamiento JSR 269 (Java SE 6+)” con una estructura muy similar al dedicado al procesamiento J2SE 1.5.

12.- COMMON ANNOTATIONS (JSR 250).

Además de las anotaciones y meta-anotaciones predefinidas básicas, se añadirían posteriormente algunas anotaciones predefinidas más: las llamadas “anotaciones comunes”, incluidas en la especificación **JSR 250 (Common Annotations for the Java™ Platform)**.

Las “anotaciones comunes” eran tipos anotación de uso común que los diseñadores del lenguaje Java crearon para que fueran adoptadas de forma generalizada por la comunidad de desarrolladores, antes de que cada uno individualmente inventara los suyos propios, dando lugar a una gran multiplicidad de tipos anotación similares para un mismo fin. Así, se pretendía evitar que las aplicaciones definieran sus propios tipos anotación si podían utilizar uno de los tipos anotación comunes estándar ya disponibles. Por tanto, el principal objetivo de las “anotaciones comunes” era **servir como referencia para evitar duplicidades** y, de esta manera, no estar reinventando la rueda continuamente, algo que desaconseja un principio básico del diseño software como es el [DRY](#) (Don't Repeat Yourself).

Otro objetivo de las “anotaciones comunes” era **posibilitar un estilo más declarativo** de programación que se sabía que sería implementado en muchas de las tecnologías de Java EE, así como mantener la consistencia entre las diferentes especificaciones agrupadas bajo el paraguas de Java EE y Java SE.

Así pues, la **JSR 250**, a diferencia de otras especificaciones, definía tipos anotación de uso común tanto para Java Standard Edition (Java SE) como para Java Enterprise Edition (Java EE). Además, a lo largo de los últimos años, se han realizado 2 revisiones de la especificación JSR 250 original, dando lugar a las revisiones Common Annotations 1.1 [[cambios](#)] y Common Annotations 1.2 [[cambios](#)]. Estas revisiones no sólo cambiaron algunos detalles del diseño y la firma de los tipos anotación comunes a lo largo del tiempo, si no que introdujeron nuevos tipos anotación comunes. Para más detalles, puede consultarse la [web oficial de la JSR 250](#).

Evolución de la especificación JSR 250		
Versión (Fecha)	Tipos Anotación Comunes definidos	
Common Annotations 1.0 (11/Mayo/2006)	Java SE 6	Java EE 5
	javax.annotation.Generated	javax.annotation.security.DeclareRoles
	javax.annotation.Resource	javax.annotation.security.RolesAllowed
	javax.annotation.Resources	javax.annotation.security.PermitAll
	javax.annotation.PostConstruct	javax.annotation.security.DenyAll
Common Annotations 1.1 (10/Dic/2009)	javax.annotation.PreDestroy	javax.annotation.security.RunAs
	Java EE 6	
	javax.annotation.ManagedBean	
Common Annotations 1.2 (14/Junio/2013)	javax.annotation.sql.DataSourceDefinition	
	javax.annotation.sql.DataSourceDefinitions	Java EE 7
	javax.annotation.Priority	

12.1.- Common Annotations 1.0 (Java SE 6, Java EE 5).

Los primeros tipos anotación de “anotaciones comunes” que definía la versión 1.0 de la especificación JSR 250 fueron introducidos en Java SE 6 en el paquete javax.annotation y en Java EE 5 en el paquete javax.annotation.security.

A continuación sigue una breve descripción de dichos “tipos anotación comunes”, para pasar en las siguientes páginas a dedicar un apartado a examinar cada una de ellas en detalle:

Commons Annotations 1.0 Tipos Anotación Comunes para Java SE 6	
Anotación	Descripción
<code>@Generated</code>	Tipo anotación que permite marcar código fuente Java como código generado por herramientas de generación automática de código.
<code>@Resource</code>	Declara una referencia a un recurso de la aplicación previamente declarado en el fichero de configuración de dicha aplicación.
<code>@Resources</code>	Tipo anotación contenedor para meter múltiples anotaciones Resource.
<code>@PostConstruct</code>	Marca los métodos que hay que ejecutar tras la construcción de una instancia de un objeto de la clase por parte del contenedor Java EE para llevar a cabo cualquier tarea de inicialización (como pueden ser la apertura de ficheros o apertura de una conexión a BD, etcétera).
<code>@PreDestroy</code>	Marca los métodos que hay que ejecutar antes de que la instancia de un objeto sea eliminada del contenedor Java EE para llevar a cabo tareas de liberación de recursos (cierre de ficheros, desconexión de BD, etc).

Commons Annotations 1.0 Tipos Anotación Comunes para Java EE 5	
Anotación	Descripción
<code>@DeclareRoles</code>	Declara en la aplicación todos los roles disponibles sobre una clase.
<code>@RolesAllowed</code>	Especifica la lista de roles permitidos en el acceso a ciertos elementos.
<code>@PermitAll</code>	Permite que todos los roles declarados puedan acceder a un elemento.
<code>@DenyAll</code>	Impide que todos los roles declarados puedan acceder a un elemento.
<code>@RunAs</code>	Define el rol de la aplicación en su ejecución en el contenedor Java EE.

12.1.1.- Common Annotations 1.0 para Java SE 6.

12.1.1.1.- @Generated

@Generated	
Descripción	Anotación que permite marcar código generado por una herramienta de generación de código.
Clase	<code>javax.annotation.Generated</code>
Target	<code>PACKAGE, TYPE, ANNOTATION_TYPE, METHOD, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, PARAMETER</code>
Retention	<code>SOURCE</code>
Otros	<code>@Documented</code>
Desde	Java SE 6, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@Generated(value = "es.uma.lcc.PFCAnotaciones.ClaseGeneradoraCodigo", comments = "Este código está generado automáticamente", date = "2014-04-23T15:30:56.235-0100") // fecha formato ISO 8601</code>
Elementos	<code>value</code> = Clase generadora del código. Nombre completamente cualificado, incluyendo paquete. <code>comments</code> = Comentario libre que puede incluir el generador de código a voluntad. <code>date</code> = Fecha del momento en que el código fue generado en formato ISO 8601.

La anotación **@Generated** indica que el elemento de código anotado por ella ha sido generado por alguna herramienta de generación de código fuente.

@Generated permite así distinguir entre código fuente que haya sido escrito por desarrolladores humanos y aquel que ha sido generado como parte de algún proceso generación de código y es por tanto creado por un algoritmo de generación automática, ya sea entre código dentro de un mismo fichero o entre diferentes ficheros de código fuente.

Por su propia naturaleza, el tipo anotación **@Generated** tiene como “target” todos los posibles tipos de elementos del lenguaje de programación Java, desde paquetes, clases, métodos, hasta variables locales. Todo lo que pueda ser escrito como elemento de código en Java puede anotarse con **@Generated**, sin ninguna excepción.

En cuanto a sus elementos: `value` es el único elemento requerido. Deberá guardar el nombre del motor generador de código que ha generado el elemento anotado. Por convención se espera que en lugar de un nombre comercial, se informe este elemento con el nombre completamente cualificado (incluyendo el paquete) de la clase que ha generado el código.

Los elementos `comments` y `date` son opcionales. En `comments` se puede guardar cualquier tipo de comentario libre que estimen oportuno los diseñadores del generador de código. En `date` se guarda la fecha en que el elemento anotado fue generado, pero sin dejar tanta libertad, ya que, si se informa el valor de este elemento, la convención a seguir según la documentación dice que la fecha debería formatearse siguiendo el formato dado por el estándar ISO 8601.

IMPORTANTE: Desde Java SE 9+ esta anotación se considera deprecada y se recomienda usar la anotación **@Generated** homónima del paquete `javax.annotation.processing`.

Ejemplo: @Generated sobre un método setter generado automáticamente

Se podría emplear el tipo anotación **@Generated** por ejemplo para anotar métodos de acceso getter, o setter, como es el caso del ejemplo mostrado, que hayan sido generados de forma automática. **NOTA:** El entorno de desarrollo Eclipse, así como otros, incluyen la posibilidad de generar código de este tipo, pero, a decir verdad, no suelen seguir la convención de anotar dicho código generado automáticamente con **@Generated**.

```
@Generated(  
    value = "es.uma.lcc.PFCAnotaciones.ClaseGeneradoraCodigo",  
    comments = "Método setter para 'propiedad1' generado automáticamente",  
    date = "2014-04-24T13:45:12.235-0100" // fecha formato ISO 8601  
)  
public void setterPropiedad1GeneradoAutomaticamente(String valorPropiedad1) {  
  
    // ... código del método generado automáticamente ...  
    this.propiedad1 = valorPropiedad1;  
}
```

12.1.1.2.- @Resource

@Resource	
Descripción	Declara una referencia a un recurso de la aplicación declarado en el contenedor Java EE.
Clase	javax.annotation.Resource
Target	FIELD, METHOD, TYPE
Retention	RUNTIME
Otros	N/D
Desde	Java SE 6, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<pre>@Resource(name = "nombreRecurso", mappedName = "nombreMapeadoRecurso", lookup = "nombreReferenciaRecurso", type = es.uma.lcc.PFCAnotaciones.ClaseRecurso, shareable = false, authenticationType = Resource.AuthenticationType.CONTAINER, description = "Descripción del recurso")</pre>
Elementos	<p>name = Nombre JNDI del recurso. Por defecto: Campos = Nombre, Métodos = Propiedad Bean. mappedName = Nombre mapeado del recurso. Específico de cada servidor. Uso no recomendado. lookup = Nombre global JNDI de un recurso compatible al que la referencia apunta. type = Clase recurso. Por defecto: Campos = Clase Campo, Métodos = Clase Propiedad Bean. shareable = Indica si el recurso es un recurso compartido con otros componentes. authenticationType = Tipo autenticación. Sólo se especifica para factorías de conexiones. description = Descripción del recurso en cuestión para una mejor información de despliegue.</p>

La anotación **@Resource** declara una referencia a un recurso debidamente configurado de la aplicación. El contenedor Java EE se encargará de buscar entre los recursos definidos en los ficheros de configuración, instanciar el recurso apropiado e injectarlo durante la ejecución del código en el momento en que sea necesario para su utilización.

El momento en el que se instancia e injecta el recurso de la aplicación por parte del contenedor Java EE varía en función de que el elemento anotado por **@Resource** sea un campo o un método (siendo en ambos casos en tiempo de inicialización), o sea una clase (en cuyo caso el recurso se injecta en tiempo de ejecución).

Tipos de inyección de dependencias en el uso de @Resource

Es importante distinguir claramente el elemento de código que aparece anotado con **@Resource**, ya que el contenedor Java EE se comporta de forma distinta según sea el caso. Como ya hemos comentado, el momento de inyección de los recursos cambia en función de esto. Los **tipos de inyección de dependencias** son pues los siguientes:

- Inyección en campos (Field-based injection)
- Inyección en métodos (Method-based injection)
- Inyección en clases (Class-based injection)

A continuación, vamos a describir con un poco más de detalle el uso de los elementos propios del tipo anotación **@Resource** y luego expondremos las particularidades de su uso para el caso de que nos encontremos en cada uno de los tipos de inyección de dependencias.

Elementos de @Resource	
Elemento	Descripción
name	Nombre JNDI del recurso. Por defecto, se comporta distinto según el tipo de inyección: - Inyección en Campos: Opcional. Por defecto: Nombre del campo. - Inyección en Métodos: Opcional. Por defecto: Nombre de la propiedad Java Bean del método. - Inyección en Clases: Requerido. No hay valor por defecto, así que debe indicarse un nombre.
mappedName	Un nombre de la implementación específica (y, por tanto, no portable) del contenedor Java EE al cual el recurso en cuestión debería ser mapeado para poder ser localizado. Ya que es una característica ligada a la implementación específica se recomienda evitar su uso si es posible.
lookup	Nombre global JNDI de un recurso compatible al que la referencia apunta.
type	El tipo (clase Java) del recurso. El valor de type se infiere automáticamente en algunos casos: - Inyección en Campos: Clase del campo anotado. - Inyección en Métodos: Clase de la propiedad Java Bean asociada al método anotado. - Inyección en Clases: No se infiere automáticamente, se tiene que establecer un valor.
shareable	Indica si el recurso será compartido por otros componentes. Por defecto su valor es false . Sólo tiene sentido establecer esta propiedad para objetos que tengan sentido que sean “compartibles”, como factorías de conexiones o recursos ORB (Object Resource Broker).
authenticationType	Tipo de autenticación de acceso al recurso. Sólo puede tener dos valores pertenecientes al tipo enumerado AuthenticationType , que son: CNTAINER (por defecto) y APPLICATION . Sólo tiene sentido en factorías de conexiones, también llamadas adaptadores o <i>data sources</i> .
description	Descripción del recurso, típicamente en el lenguaje por defecto en el cual la aplicación es desplegada. Esta información se utilizará para facilitar la identificación de recursos y ayudar a los desarrolladores y a los desplegadores de la aplicación a configurar los recursos adecuados.

Inyección en campos con @Resource

La inyección en campos ocurre cuando se declara un campo en una clase Java y se anota con el tipo anotación `@Resource`. El contenedor inferirá el nombre y el tipo del recurso si no se establecen los valores de los elementos `name` y `type`. Si se especificara el tipo, este deberá coincidir con el tipo (clase Java) de la declaración del campo.

Inferencia de valores de los elementos de @Resource por parte del contenedor Java EE	
Código	Valores inferidos por el contenedor
<pre>package es.uma.lcc.PFCAnotaciones; public class ClaseEjemplo { @Resource private javax.sql.DataSource miBD; }</pre>	<code>name</code> = es.uma.lcc.PFCAnotaciones/miBD <code>type</code> = javax.sql.DataSource.class
<pre>package es.uma.lcc.PFCAnotaciones; public class ClaseEjemplo { @Resource(name="nombreBaseDatos") private javax.sql.DataSource miBD; }</pre>	<code>name</code> = nombreBaseDatos <code>type</code> = javax.sql.DataSource.class

Inyección en métodos con @Resource

La inyección en métodos ocurre cuando se declara un método en una clase Java y se anota con el tipo anotación `@Resource`. El contenedor inferirá el nombre y el tipo del recurso si no se establecen los valores de los elementos `name` y `type`. Para ello, el método setter deberá seguir las reglas de Java Beans en cuanto a los nombres de las propiedades: el nombre del método deberá comenzar con `set` y tener un tipo de retorno `void`, así como un único parámetro. Si se especificara el tipo, este deberá coincidir con el tipo (clase Java) de la propiedad.

Inferencia de valores de los elementos de @Resource por parte del contenedor Java EE	
Código	Valores inferidos por el contenedor
<pre>package es.uma.lcc.PFCAnotaciones.ejemplos; public class ClaseEjemplo { private javax.sql.DataSource miBD; @Resource private void setMiBD(javax.sql.DataSource ds) { miBD = ds; } }</pre>	<code>name</code> = es.uma.lcc.PFCAnotaciones/miBD <code>type</code> = javax.sql.DataSource.class
<pre>package es.uma.lcc.PFCAnotaciones.ejemplos; public class ClaseEjemplo { private javax.sql.DataSource miBD; @Resource(name="nombreBaseDatos") private void setMiBD(javax.sql.DataSource ds) { miBD = ds; } }</pre>	<code>name</code> = nombreBaseDatos <code>type</code> = javax.sql.DataSource.class

Inyección en clases con @Resource

La inyección en clases ocurre cuando se declara una clase Java y se anota con el tipo anotación `@Resource`. A diferencia de lo que ocurre con la inyección de campos y métodos, en la inyección en clases, el contenedor no inferirá por defecto el nombre y el tipo del recurso si no se establecen los valores de los elementos `name` y `type`. Por ello, es obligatorio establecer sus valores siempre.

```
@Resource (name="colaMensajes", type="javax.jms.ConnectionFactory")
public class MensajeBeanEjemplo {
    // ... código de la clase ...
}
```

12.1.1.3.- @Resources

@Resources	
Descripción	Tipo anotación contenedor que permite agrupar en un array múltiples anotaciones <code>@Resource</code> .
Clase	<code>javax.annotation.Resources</code>
Target	<code>TYPE</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 6, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@Resources ({ @Resource (name="colaMensajes", type="javax.jms.ConnectionFactory"), @Resource (name="sesionJavaMail", type="javax.mail.Session") })</code>
Elementos	<code>value = Resource[]</code> . Array de <code>Resource</code> , para poder declarar múltiples recursos de una vez.

La anotación **@Resources** es una forma de agrupar la declaración de múltiples recursos cuando la anotación se realiza sobre una clase (**@Resources** no se aplica a campos o métodos, a diferencia de **@Resource**).

Ejemplo: @Resources definiendo múltiples recursos sobre una clase

El siguiente código declara dos recursos al mismo tiempo sobre una clase: el primer recurso es una cola de mensaje de JMS (Java Message Service) y el otro es una sesión de JavaMail, con la que podríamos consultar el estado del servidor de correo, o enviar o recibir correos electrónicos.

```
@Resources ({
    @Resource (name="colaMensajes", type="javax.jms.ConnectionFactory"),
    @Resource (name="sesionJavaMail", type="javax.mail.Session")
})
public class MensajeBeanEjemplo {
    // ... código de la clase ...
}
```

12.1.1.4.- @PostConstruct

@PostConstruct	
Descripción	Marca el método que hay que ejecutar tras la construcción de una instancia de una clase.
Clase	<code>javax.annotation.PostConstruct</code>
Target	<code>METHOD</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 6, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@PostConstruct</code>
Elementos	N/D (anotación marcadora)

La anotación **@PostConstruct** marca el método que hay que ejecutar tras la construcción de una instancia de un objeto de la clase por parte del contenedor Java EE para llevar a cabo cualquier tarea de inicialización (como pueden ser la apertura de ficheros o apertura de una conexión a BD, entre otras).

Los componentes de la aplicación son construidos por el contenedor Java EE en el cual son desplegados. Si es necesario inicializar dichos componentes a valores distintos de los valores por defecto, o si hay que realizar alguna operación de inicialización de algún tipo de recurso (como ficheros, conexiones a BD, etcétera), puede crearse un método de inicialización y anotarlo con **@PostConstruct**, lo cual provocará que dicho método se ejecute justo a continuación de completar la instanciación (creación) de un objeto, mediante el constructor correspondiente, y tras la realización de inyección de dependencias (si procede). El método de ejecutará para todo nuevo objeto instanciado de la clase.

Hay algunas reglas específicas que deben seguirse al usar este tipo anotación:

- Sólo puede haber un único método anotado con **@PostConstruct** para cada clase.
- Debe retornar `void` y no ser `static`.
- No debe tener parámetros (salvo para interceptores EJB, en cuyo caso se pasaría un contexto).
- No debe arrojar excepciones comprobadas (aquellas que no heredan de `RuntimeException`).
- Puede tener cualquier visibilidad y ser o no ser marcado como `final`.

Ejemplo: @ PostConstruct sobre un método de inicialización de recursos

El siguiente ejemplo hace que se ejecute el método etiquetado cada vez que se instancie un objeto de la clase en el que esté definido.

```
@PostConstruct
public void inicializarConexionBaseDatos() {
    // ... código de inicialización de base de datos...
}
```

12.1.1.5.- @PreDestroy

@PreDestroy	
Descripción	Marca el método que hay que ejecutar antes de la destrucción de una instancia de una clase.
Clase	<code>javax.annotation.PreDestroy</code>
Target	<code>METHOD</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java SE 6, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@PreDestroy</code>
Elementos	N/D (anotación marcadora)

La anotación **@PreDestroy** marca los métodos que hay que ejecutar antes de la destrucción de una instancia de un objeto de la clase por parte del contenedor Java EE para llevar a cabo cualquier tarea de liberación de recursos (cierre de ficheros, cierre de conexiones a BD, entre otras). Es la anotación “complementaria” de **@PostConstruct**.

Los componentes de la aplicación son destruidos por el contenedor Java EE en el cual son desplegados. Si es necesario liberar recursos asociados a ellos, puede crearse un método de liberación de recursos y anotarlo con **@PreDestroy**, lo cual provocará que dicho método se ejecute antes de finalizar la destrucción del objeto. El método se ejecutará para todo objeto instanciado de la clase que vaya a ser destruido.

Hay algunas reglas específicas que deben seguirse al usar este tipo anotación (que son básicamente las mismas que para el tipo anotación **@PostConstruct**):

- Sólo puede haber un único método anotado con **@PreDestroy** para cada clase.
- Debe retornar `void` y no ser `static`.
- No debe tener parámetros (salvo para interceptores EJB, en cuyo caso se pasaría un contexto).
- No debe arrojar excepciones comprobadas (aquellas que no heredan de `RuntimeException`).
- Puede tener cualquier visibilidad y ser o no ser marcado como `final`.

Ejemplo: @PreDestroy sobre un método de liberación de recursos

El siguiente ejemplo hace que se ejecute el método etiquetado cada vez que se vaya a destruir un objeto de la clase en el que esté definido.

```
@PreDestroy
public void liberarConexionBaseDatos() {
    // ... código de cierre y liberación de recursos de la conexión a base de datos...
}
```

12.1.2.- Common Annotations 1.0 para Java EE 5.

12.1.2.1.- @DeclareRoles

@DeclareRoles	
Descripción	Anotación que permite declarar la lista de roles válidos de acceso a una determinada clase.
Clase	<code>javax.annotation.security.DeclareRoles</code>
Target	<code>TYPE</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java EE 5, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@DeclareRoles({ "NombreRol1", "NombreRol2", ..., "NombreRolN" })</code>
Elementos	<code>value = String[]</code> . Array de nombres de roles válidos para acceder al código de la clase.

La anotación **@DeclareRoles** permite declarar una lista de nombres de roles de seguridad válidos para el acceso al código de una determinada clase.

La anotación **@DeclareRoles** sólo declara los nombres de los roles, por lo que será responsabilidad del desarrollador proporcionar el código que implemente el control de acceso en base a los roles válidos declarados, ya sea directamente o utilizando el tipo anotación relacionado **@RolesAllowed**.

Ejemplo: @DeclareRoles y código que realiza el control de acceso

En el siguiente ejemplo se utiliza la anotación **@DeclareRoles** para indicar los roles válidos de acceso a una clase. No obstante, la anotación por sí sola no implementa el control del correcto acceso al código de la clase, por lo que se implementa un método `rolPermitido` que, a través de la información de contexto que proporciona el contenedor Java EE, permite comprobar si el código llamante pertenece a un rol válido o no.

```
@DeclareRoles({ "Director", "Manager", "Otros" })
public class ClaseEjemplo{

    @Resource
    private Context contexto;

    private boolean rolPermitido(){
        return contexto.isCallerInRole("Director")
            || contexto.isCallerInRole("Manager")
            || contexto.isCallerInRole("Otros");
    }

    // ... resto de código que usa el metodo rolPermitido() para el control de acceso...
}
```

12.1.2.2.- @RolesAllowed

@RolesAllowed	
Descripción	Define los roles con acceso al elemento de código anotado. Control automático.
Clase	<code>javax.annotation.security.RolesAllowed</code>
Target	<code>TYPE, METHOD</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java EE 5, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@RolesAllowed({ "NombreRol1", "NombreRol2", ..., "NombreRolN" })</code>
Elementos	<code>value = String[].</code> Roles para los que se concede acceso al elemento de código anotado.

La anotación **@RolesAllowed** permite declarar una lista de nombres de roles de seguridad a los que se permite el acceso al código anotado, ya sea, en el caso de este tipo anotación, una clase o un método.

La anotación **@RolesAllowed**, a diferencia de **@DeclareRoles**, sí provoca que se realice el control automático de los roles de acceso por parte del contenedor Java EE, sin tener el desarrollador que tomarse la molestia implementar ningún mecanismo a tal efecto.

Al poder aplicarse este tipo anotación tanto a clases como a métodos, puede darse el caso de que los valores establecidos para una clase y los establecidos para un método de la clase con este mismo tipo anotación difieran. Para solventar la ambigüedad de dichos casos, existen unas **reglas de valores por defecto y precedencia para establecer los roles de acceso permitidos**, que son las siguientes:

- A nivel de Clase, Por defecto: Se toman todos los roles declarados por **@RolesDeclared**.
- A nivel de Método, Por defecto: Se toman los roles de **@RolesAllowed** de la Clase o, en caso de que la clase no esté anotada con **@RolesAllowed**, todos los declarados por **@RolesDeclared**.
- A nivel de Método, Precedencia: Si ambos, tanto la clase como el método, están anotados con **@RolesAllowed**, se toman a los valores de la anotación del método, ignorando los de la clase.

Ejemplo: @RolesAllowed y varios ejemplos de valores por defecto y precedencia

El siguiente ejemplo ilustra el uso de las anotaciones **@DeclareRoles** y **@RolesAllowed**, así como las reglas de valores por defecto y precedencia explicadas para **@RolesAllowed**.

```
@DeclareRoles({"A", "B", "C", "X", "Y", "Z"}) // lista de TODOS los roles válidos
@RolesAllowed({"A", "B", "C"}) // PERMITIDOS (a los demás se les denegará el acceso)
public class ClaseEjemplo{

    public void metodo1() {
        // roles permitidos: A, B, C
        // (valor por defecto de @RolesAllowed de la clase)
    }

    @RolesAllowed(value = {"A", "X", "Y"} )
    public void metodo2() {
        // roles permitidos: A, X, Y
        // (precedencia del valor del método)
    }
}
```

12.1.2.3.- @PermitAll

@PermitAll	
Descripción	Permite el acceso por parte de todos los roles. Desactiva el control automático.
Clase	<code>javax.annotation.security.PermitAll</code>
Target	<code>TYPE, METHOD</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java EE 5, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@PermitAll</code>
Elementos	N/D (anotación marcadora)

La anotación **@PermitAll** especifica que, ya sea una clase o un método, se permite a todos los roles invocar código dentro del elemento de código anotado.

La anotación **@PermitAll**, a efectos prácticos, viene a ser como si se desactivara el control automático llevado a cabo por el contenedor Java EE sobre el código anotado con dicho tipo anotación, pasando a ser código “no seguro” o, al menos, “no comprobado”.

Para este tipo de anotación, al ser aplicable tanto a clases como métodos y poder existir la posibilidad de que unos elementos de código subsuman a otros, existe una regla especial que se aplica en dichos casos y que podríamos llamar “**regla de inclusión**”: y es que si, por ejemplo, se anota una clase con `@PermitAll`, todos los métodos de dicha clase, aunque tengan unos valores de roles permitidos restringidos por `@RolesAllowed`, en esta caso no se aplica la regla habitual de precedencia más interior, si no que `@PermitAll` hace que el método sea accesible por todos los roles, estén declarados/permitidos o no.

Ejemplo: @PermitAll y ejemplos de la regla de inclusión

```
@DeclareRoles({"A", "B", "C", "X", "Y", "Z"})
@RolesAllowed({"A", "B", "C"})
@PermitAll // TODOS LOS ROLES PERMITIDOS
public class ClaseEjemplo{

    public void metodo1() {
        // roles permitidos: TODOS
        // (no solo los declarados por @DeclareRoles, cualquiera puede invocar este método)
        // (ante @PermitAll se ignora el valor por defecto de @RolesAllowed de la clase)
    }

    @RolesAllowed(value = {"A", "X", "Y"} )
    public void metodo2() {
        // roles permitidos: TODOS
        // (no solo los declarados por @DeclareRoles, cualquiera puede invocar este método)
        // (ante @PermitAll se ignora la precedencia del valor del método)
    }
}
```

12.1.2.4.- @DenyAll

@DenyAll	
Descripción	Deniega el acceso a de todos los roles. Impide la ejecución del código anotado.
Clase	<code>javax.annotation.security.DenyAll</code>
Target	<code>TYPE, METHOD</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java EE 5, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@DenyAll</code>
Elementos	N/D (anotación marcadora)

La anotación **@DenyAll** especifica que, ya sea una clase o un método, no se permite a ningún rol invocar código dentro del elemento de código anotado.

La anotación **@DenyAll**, a efectos prácticos, impide que se ejecute código dentro del elemento anotado por parte del contenedor Java EE. Es como si el código anotado hubiera sido “desactivado” o “bloqueado”.

Para este tipo anotación, al ser aplicable tanto a clases como métodos y poder existir la posibilidad de que unos elementos de código subsuman a otros, existe una regla especial que se aplica en dichos casos y que podríamos llamar “**regla de exclusión**”: y es que si, por ejemplo, se anota una clase con `@DenyAll`, todos los métodos de dicha clase, aunque tengan unos valores de roles permitidos dados por `@RolesAllowed`, en esta caso no se aplica la regla habitual de precedencia más interior, si no que `@DenyAll` hace que el método sea inaccesible para todos los roles, estén declarados/permitidos o no.

Ejemplo: @DenyAll y ejemplos de la regla de exclusión

```
@DeclareRoles({"A", "B", "C", "X", "Y", "Z"})
@RolesAllowed({"A", "B", "C"})
@DenyAll // TODOS LOS ROLES DENEGADOS
public class ClaseEjemplo{

    public void metodo1() {
        // roles permitidos: NINGUNO
        // (ni los declarados por @DeclareRoles, ni nadie puede invocar este método)
        // (ante @DenyAll se ignora el valor por defecto de @RolesAllowed de la clase)
    }

    @RolesAllowed(value = {"A", "X", "Y"} )
    public void metodo2() {
        // roles permitidos: NINGUNO
        // (ni los declarados por @DeclareRoles, ni nadie puede invocar este método)
        // (ante @DenyAll se ignora la precedencia del valor del método)
    }
}
```

12.1.2.4.- @RunAs

@RunAs	
Descripción	Establece el rol durante la ejecución del código de la clase anotada en el contenedor Java EE.
Clase	<code>javax.annotation.security.RunAs</code>
Target	<code>TYPE</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java EE 5, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@RunAs</code>
Elementos	<code>value = String</code> . Nombre del rol con el que se ejecutará el código de la clase anotada.

La anotación **@RunAs** define la identidad de la aplicación durante la ejecución de su código en el contenedor Java EE.

El principal interés de la anotación **@RunAs**, a efectos prácticos, es el de facilitar labores de depuración, ya que permite a los desarrolladores ejecutar código bajo un determinado rol.

El nombre del rol que se establece como `value` de este tipo anotación debe coincidir con la información de usuarios y grupos del dominio de seguridad que se haya configurado para el contenedor Java EE.

Ejemplo: @RunAs y un ejemplo típico de depuración usando un rol específico

Lo siguiente es un ejemplo de como los desarrolladores pueden usar el tipo anotación **@RunAs** para ejecutar código simulando ostentar un determinado rol de seguridad. De esta forma, pueden reproducir el comportamiento del método en unas circunstancias exactas a aquellas en las cuales el método se ejecutará finalmente, puesto que los desarrolladores en la versión final de la aplicación obviamente no deberán poder tener acceso a la aplicación con el rol “Director”.

```
public class ClaseEjemplo{  
  
    @RunAs("Director")  
    public void metodoParaDirectores() {  
        // usando RunAs podemos ejecutar el código de este método  
        // con el rol de "Director", aunque nuestro rol no sea ese  
        // en realidad, pero esto nos permite depurar su código al  
        // reproducir las condiciones en que dicho método se ejecuta  
    }  
}
```

12.2.- Common Annotations 1.1 (Java EE 6).

12.2.1.- @ManagedBean

@ManagedBean	
Descripción	Marca un POJO como ManagedBean (un JavaBean que soporta servicios especiales de Java EE).
Clase	<code>javax.annotation.ManagedBean</code>
Target	<code>TYPE</code>
Retention	<code>RUNTIME</code>
Otros	N/D
Desde	Java EE 6, JSR 250 (Common Annotations for the Java™ Platform) versión 1.1
Signatura	<code>@ManagedBean</code>
Elementos	<code>value = String</code> . Nombre del ManagedBean. Normalmente no se informa.

La anotación **@ManagedBean** marca un [POJO](#) (Plain Old Java Object, es decir, un objeto Java tradicional) como un ManagedBean, que soporta servicios especiales de Java EE.

Los ManagedBean, también llamados “componentes ligeros” (“*lightweight components*”) fueron introducidos en una especificación propia que formaba una parte de la especificación global [JSR 316](#) (Java EE 6).

Los ManagedBean son Java Beans cuyo ciclo de vida es gobernado por el contenedor Java EE, permitiendo de esta manera *callbacks* de creación y destrucción, así como soporte de inyección de recursos (además de poder ser inyectados ellos mismos). De esta forma, los ManagedBean se constituyen como un nuevo y más potente tipo de Java Bean para la plataforma de desarrollo Java EE que soporta nuevos servicios de última generación, como inyección, ciclo de vida, *callbacks* e interceptores.

Para definir un ManagedBean, simplemente hay que anotar una clase con el tipo anotación **@ManagedBean** y ya dispondremos de todas las ventajas de este tipo de nuevos objetos Java Bean tan potentes y versátiles.

Ejemplo:

Lo siguiente es un ejemplo donde la clase anotada será manejada por el contenedor Java EE como un ManagedBean gracias a estar anotada con el tipo anotación **@ManagedBean**. Podemos ver también en el ejemplo otra capacidad de los ManagedBean: la inyección, ya que en esta clase se inyecta otro ManagedBean como recurso (la clase `LocalizadorBean` suponemos que está también anotada con `@ManagedBean`) gracias al tipo anotación `@Resource`.

```
@ManagedBean
public class LocalizacionTextosBean {
    @Resource
    LocalizadorBean localizador;

    public String obtenerTextoLocalizado(String claveTexto) {
        return localizador.traducir(claveTexto);
    }
}
```

12.2.2.- @DataSourceDefinition

@DataSourceDefinition	
Descripción	Define un DataSource (origen de datos) para la aplicación dentro del contenedor Java EE.
Clase	<code>javax.annotation.sql.DataSourceDefinition</code>
Target	TYPE
Retention	RUNTIME
Otros	N/D
Desde	Java EE 6, JSR 250 (Common Annotations for the Java™ Platform) versión 1.1
Signatura	<pre> @DataSourceDefinition (className="paquete.ClaseImplementacionDataSource", name="java:global/jdbc/nombreDataSource", url="jdbc:proveedorBD://servidorBD:1234/nombreBD;user=usuario", serverName="servidorBD", portNumber=1234, user="usuario", password="password", databaseName="nombreBD", description="Texto descriptivo del origen de datos.", initialPoolSize=5, isolationLevel=Connection.TRANSACTION_READ_COMMITTED, loginTimeout=30, maxIdleTime=600, minPoolSize=2, maxPoolSize=10, maxStatements=20, transactional=true, properties={"prop1=val1", "prop2=val2", ..., "propN=valN"}) </pre>
Elementos	<p>className = Nombre cualificado de la clase Java que implementa el DataSource.</p> <p>name = Nombre JNDI del DataSource para su posterior localización en el contenedor Java EE.</p> <p>url = URL JDBC. Los parám. puestos aquí no deben ponerse de nuevo en los demás elementos.</p> <p>serverName = Nombre de red o dirección IP del servidor de BD.</p> <p>portNumber = Número del puerto de conexión al servidor de BD.</p> <p>user = Usuario de conexión al servidor de BD.</p> <p>password = Contraseña de conexión al servidor de BD.</p> <p>databaseName = Nombre de la BD dentro del servidor de BD.</p> <p>description = Texto descriptivo del origen de datos: entorno, versión, propósito, etcétera.</p> <p>initialPoolSize = Número de conexiones cuando se inicialice el pool de conexiones.</p> <p>isolationLevel = Nivel de aislamiento de las conexiones. Valores válidos: enum Connection.</p> <p>loginTimeout = Timeout de conexión a BD (seg). 0 = timeout por defecto / sin timeout.</p> <p>maxIdleTime = Tiempo máximo (seg) para mantener una conexión ociosa abierta en el pool.</p> <p>minPoolSize = Mínimo de conexiones simultáneas abiertas del pool.</p> <p>maxPoolSize = Máximo de conexiones simultáneas abiertas del pool.</p> <p>maxStatements = Máximo de sentencias que el pool debería mantener abiertas. 0 = sin caché.</p> <p>transactional = true para conex. transaccionales. false para no participar de transacciones.</p> <p>properties = String[]. Array de propiedades no estándar y específicas de BD particulares y menos habituales (como dataSourceName, networkProtocol, propertyCycle, roleName, etcétera). Cada propiedad se especificará con el siguiente formato: "nombrePropiedad=valorPropiedad". Además, si en properties se define una propiedad que ya está definida como en su elemento propio, el valor dado por el elemento toma precedencia. Ejemplo: si definimos user="usr1" y en properties establecemos "user=usr2", se tomará el usuario del elemento user: usr1.</p>

La anotación **@DataSourceDefinition** permite definir un DataSource (origen de datos), esto es: una configuración de conexión a una base de datos, para el uso de la aplicación dentro del contenedor Java EE.

La anotación **@DataSourceDefinition** ha sido un paso extremadamente importante en la mejora, estandarización y simplificación del proceso de configuración de las aplicaciones Java EE, ya que, antes de que existiera este tipo anotación, la forma de configurar los DataSource no estaba estandarizada y era específica de cada servidor de aplicaciones Java EE (típicamente cada uno lo hacía en un fichero XML distinto), lo cual daba lugar a muchos problemas de despliegue a los desarrolladores de aplicaciones, además de complicar bastante la portabilidad de un servidor de aplicaciones Java EE a otro. Ahora, usando **@DataSourceDefinition**, sabemos que los DataSource que definamos serán entendibles por cualquier servidor con Java EE 6.

En el apartado dedicado al elemento `properties` del cuadro resumen del tipo anotación, se comenta que si a una propiedad se le da valor en un elemento propio así como dentro de una de las propiedades de `properties`, precede el valor del elemento específico de dicha propiedad. A diferencia de lo anterior, si una propiedad se establece a través del elemento `url` y también a través de otro elemento concreto, la especificación no fija cuál debería ser la precedencia en dicho caso, por lo que hay que tratar de nunca repetir la misma propiedad en más de un elemento de **@DataSourceDefinition**. Véase el siguiente ejemplo:

```
@DataSourceDefinition(
    name="java:global/MyApp/MyDataSource",
    className="org.apache.derby.jdbc.ClientDataSource",
    url="jdbc:derby://localhost:1527/myDB;user=bill",
    user="lance",
    password="secret",
    databaseName="testDB",
    serverName="luckydog"
)
```

En este caso, la especificación no fija cual de ambos valores de la propiedad `user` debería tener precedencia, si el valor de `url`: `bill`, o el valor del elemento `user`: `lance`. Debido a esto, se recomienda que las propiedades de un DataSource nunca aparezcan más de una vez en la definición del DataSource dada por el tipo anotación **@DataSourceDefinition**.

Ejemplo: **@DataSourceDefinition** definiendo un acceso a BD y su inyección

El siguiente ejemplo muestra cómo se puede definir un DataSource y cómo posteriormente se puede utilizar el tipo anotación **@Resource** para inyectarlo en una propiedad de la clase cliente utilizando el nombre JNDI declarado en la definición del DataSource para referenciarlo.

```
@DataSourceDefinition(
    name="java:app/env/inventario",
    className="oracle.jdbc.pool.OracleDataSource",
    serverName="atlantis",
    portNumber=1521,
    databaseName="inventario",
    user="miguel",
    password="abracadabra",
    properties={"driverType=thin"}
)
@Stateless
public class InventarioBean {
    @Resource(lookup="java:app/env/inventario")
    private DataSource inventarioDataSource;
    // ... resto del código de la clase donde se usa el DataSource...
}
```

12.2.3.- @DataSourceDefinitions

@DataSourceDefinitions	
Descripción	Tipo anotación contenedor que permite agrupar en un array múltiples @DataSourceDefinition.
Clase	<code>javax.annotation.sql.DataSourceDefinitions</code>
Target	<code>TYPE</code>
Retention	<code>RUNTIME</code>
Otros	N/D
Desde	Java SE 6, JSR 250 (Common Annotations for the Java™ Platform) versión 1.0
Signatura	<code>@DataSourceDefinitions({ @DataSourceDefinition(name="dataSource1", ...), @DataSourceDefinition(name="dataSource2", ...), ..., @DataSourceDefinition(name="dataSourceN", ...) })</code>
Elementos	<code>value = DataSourceDefinition[]. Array de múltiples DataSourceDefinition.</code>

La anotación **@DataSourceDefinitions** es una forma de agrupar la definición de múltiples DataSources sobre una clase, sirviendo de contenedor para múltiples tipos anotación **@DataSourceDefinition** separados por comas, tal y como se esquematiza en la signatura de ejemplo del cuadro resumen de referencia.

Ejemplo: @DataSourceDefinitions definiendo múltiples DataSource

El siguiente ejemplo declara vía un tipo anotación contenedor **@DataSourceDefinitions** un par de DataSources usando dos anotaciones **@DataSourceDefinition**. Como podemos ver, se trata de dos bases de datos de prueba diferentes, que incluso corresponden a sistemas gestores de bases de datos diferentes (una es una BD de [Apache Derby](#) y la otra una BD de [MySQL](#)). Ambos DataSource además estarán disponibles a través de sus nombres JNDI como recursos para poder ser injectados allí donde se necesiten, de forma que se puedan llevar a cabo las pruebas de la aplicación con una u otra BD según las necesidades del proceso de desarrollo.

```
@DataSourceDefinitions({  
    @DataSourceDefinition(name = "java:global/MiAplicacion/miDataSource1",  
        className = "org.apache.derby.jdbc.ClientDataSource",  
        portNumber = 1527,  
        serverName = "localhost",  
        databaseName = "baseDatosPrueba1",  
        user = "miguel",  
        password = "abracadabra",  
        properties = {"createDatabase=create"}),  
    @DataSourceDefinition(name = "java:global/MiAplicacion/miDataSource2",  
        className = "com.mysql.jdbc.jdbc2.optional.MysqlDataSource",  
        portNumber = 3306,  
        serverName = "localhost",  
        databaseName = "baseDatosPrueba2",  
        user = "angel",  
        password = "patadecabra",  
        properties = {"pedantic=true"})  
})
```

12.3.- Common Annotations 1.2 (Java EE 7).

12.3.1.- @Priority

@Priority	
Descripción	Indica la prioridad de ejecución de una clase dentro del contenedor Java EE.
Clase	<code>javax.annotation.Priority</code>
Target	<code>TYPE</code>
Retention	<code>RUNTIME</code>
Otros	<code>@Documented</code>
Desde	Java EE 7, JSR 250 (Common Annotations for the Java™ Platform) versión 1.2
Signatura	<code>@Priority(100)</code>
Elementos	<code>value = int</code> . Número que indica la prioridad. Normalmente será positivo, aunque se pueden dar el caso de valores negativos para indicar situaciones especiales o excepcionales.

La anotación **@Priority** sirve simplemente para indicar una prioridad de ejecución o de uso de una determinada clase dentro del contenedor Java EE, siguiendo algún tipo de especificación que indique cómo debe hacerse uso de dicha prioridad.

A efectos prácticos, el tipo anotación **@Priority** es algo muy sencillo. Es simplemente un tipo anotación cuyo valor es un número entero, pero la gracia está en cómo habilita un sistema de prioridades sobre el uso de las clases que otras especificaciones relacionadas con la plataforma de desarrollo Java EE pueden aprovechar. De hecho, sin ir más lejos, ya utilizan el tipo anotación **@Priority** en Java EE: los [interceptores](#), las [alternativas](#) y los [decoradores](#). Todos ellos usan la prioridad dada por el tipo anotación **@Priority** para sus diversas funciones.

En el caso de los interceptores, por ejemplo, se realiza la ejecución de los mismos de menor a mayor prioridad (orden ascendente), definiéndose además como referencia para los valores a utilizar dentro del contenedor Java EE el tipo enumerado `Interceptor.Priority`:

Tipo enumerado Interceptor.Priority		
Campo	Valor	Descripción
<code>PLATFORM_BEFORE</code>	<code>0</code>	Inicio del rango de interceptores tempranos de especificaciones de la plataforma.
<code>LIBRARY_BEFORE</code>	<code>1000</code>	Inicio del rango de interceptores tempranos de librerías de extensión.
<code>APPLICATION</code>	<code>2000</code>	Inicio del rango de interceptores definidos por las aplicaciones.
<code>LIBRARY_AFTER</code>	<code>3000</code>	Inicio del rango de interceptores tardíos de librerías de extensión.
<code>PLATFORM_AFTER</code>	<code>4000</code>	Inicio del rango de interceptores tardíos de especificaciones de la plataforma.

Ejemplo: @Priority para definir la prioridad de ejecución de un interceptor

En el siguiente ejemplo se ilustra cómo se utiliza el tipo anotación **@Priority** para definir la prioridad de ejecución de una clase definida como un interceptor Java EE. Nótese el uso del tipo enumerado `Interceptor.Priority` para marcar el valor del inicio de un rango válido y el número, en este caso 5, que se le suma a modo de “desplazamiento” (“offset”) dentro de dicho rango.

```
@Priority(Interceptor.Priority.APPLICATION + 5)
@Interceptor // anotación que describe la clase como interceptor al contenedor Java EE
public class EjemploInterceptor { ... }
```

13.- JAVA COMPILER API (JSR 199).

13.1.- Introducción.

La **Java Compiler API** ([JSR 199](#)) permite la construcción y configuración de tareas de compilación de código totalmente personalizadas en el entorno de ejecución de una aplicación Java. De esta forma, cualquier aplicación Java tendrá la posibilidad de realizar tareas de compilación, pasando a ser la dicha compilación un servicio a nivel de aplicación.

Algunos ejemplos de **usos típicos que se podrían dar a la Java Compiler API** son:

- Minimizar el tiempo de despliegue de los servidores de aplicaciones Java EE a la hora de desplegar aplicaciones, por ejemplo, evitando la ineficiencia de llamar a un compilador externo a la hora de compilar el código fuente de las páginas JSP para la generación de servlets.
- Herramientas de desarrollo como IDEs y analizadores de código pueden invocar el compilador directamente desde dentro de su instancia de aplicación y así reducir significativamente el tiempo de compilación.

Las clases relacionadas con el compilador se encuentran en el paquete `javax.tools`. La clase `ToolProvider` de este paquete proporciona el método `getSystemJavaCompiler()` que devuelve una instancia de una clase que implementa la interfaz `JavaCompiler`.

A través de una instancia `JavaCompiler` se puede realizar la compilación de dos formas:

(1) “Método clásico”: se le pasan directamente todos los argumentos en un `String[]` al método `run` del compilador y se obtiene como resultado el clásico código de retorno.

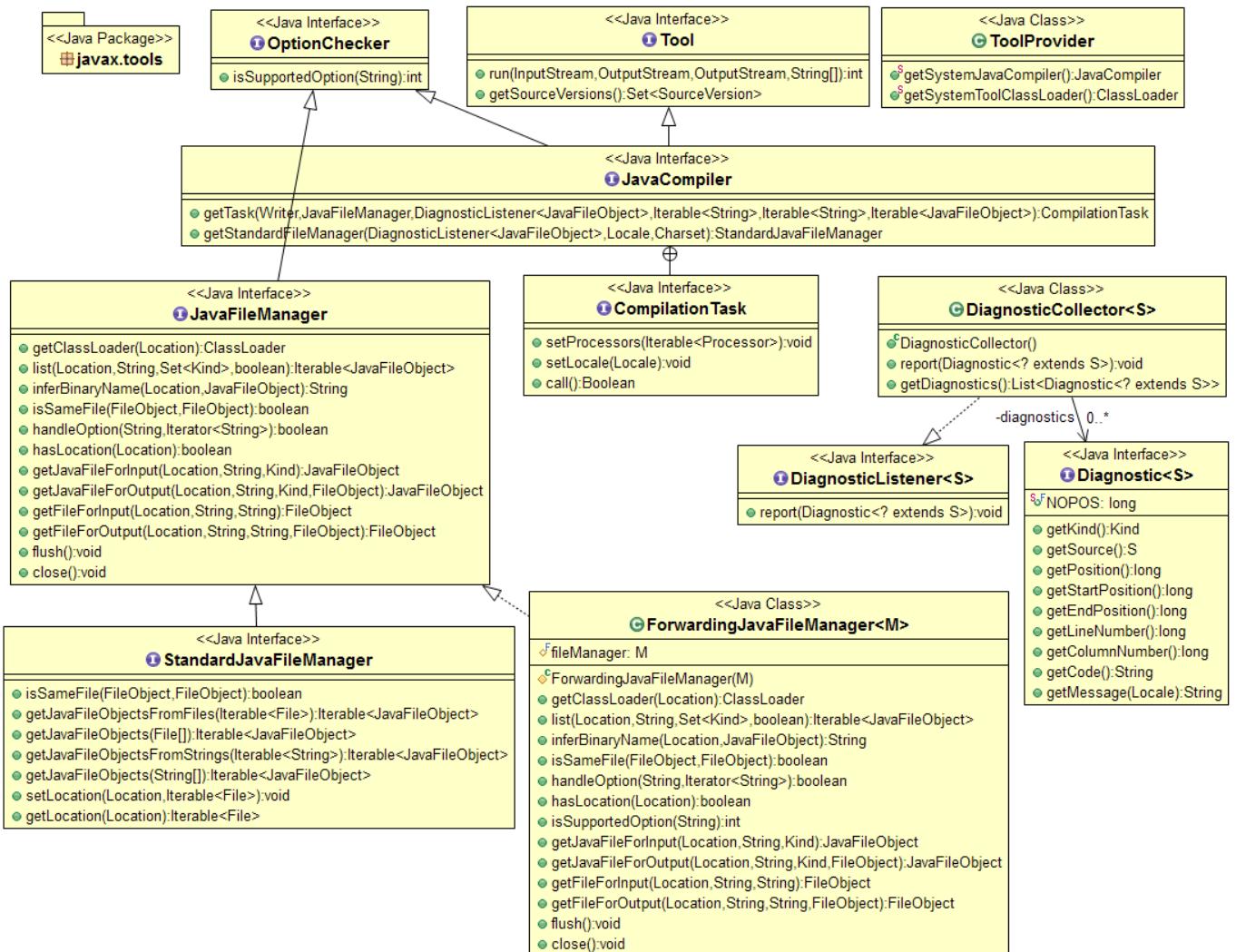
(2) “Método nuevo”: haciendo uso de las nuevas clases de la Java Compiler API que permiten crear “tareas de compilación”, se ejecutan con su método `call` y se obtiene como resultado un booleano que nos indica si la compilación ha finalizado sin errores.

Para crear una tarea de compilación (interfaz `CompilationTask`), se necesitará pasar los ficheros a ser compilados. Para esto, la Java Compiler API proporciona una abstracción de sistemas de ficheros llamada `JavaFileManager`, que permite que los ficheros se puedan recibir de distintas fuentes, como sistemas de ficheros, bases de datos, directamente de memoria, etc. `StandardFileManager` es la implementación del sistema de ficheros estándar basado en la API `java.io`. Puede obtenerse una implementación del mismo llamando al método `getStandardFileManager()` de la instancia de `JavaCompiler`.

Para ser notificado errores no fatales, al método `getStandardFileManager()` se le puede pasar un listener de diagnósticos (interfaz `DiagnosticListener`). Si se pasa un listener tendremos la posibilidad de recibir notificaciones de los mensajes emitidos durante el proceso de compilación directamente desde el compilador.

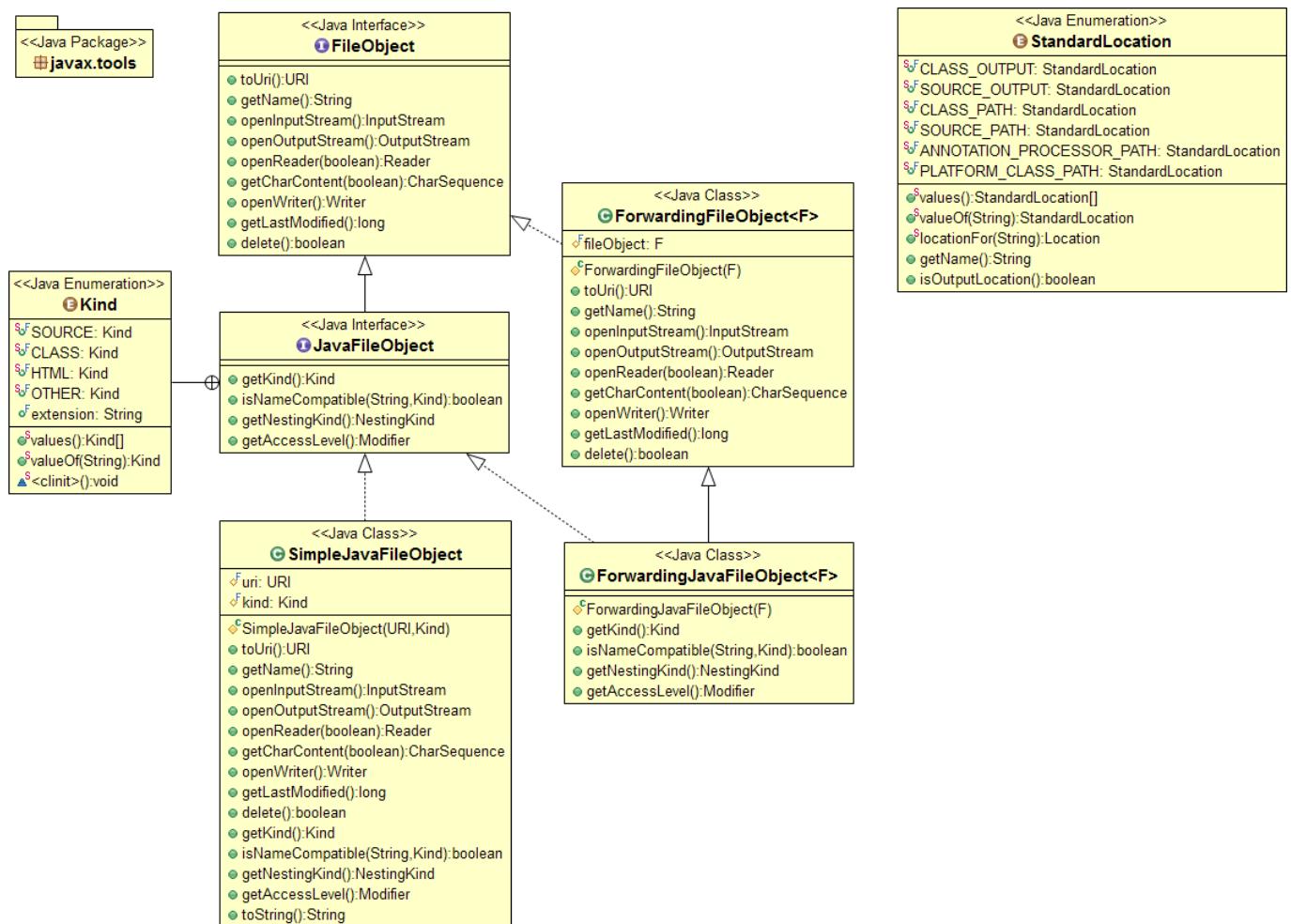
13.2.- Paquete javax.tools.

javax.tools – Compilación, Sistemas de Ficheros y Diagnósticos	
Interfaz / Clase	Descripción
JavaCompiler	Interfaz del compilador. Permite realizar la compilación directamente mediante el método <code>run</code> o mediante la creación y ejecución de tareas de compilación.
CompilationTask	Tarea de compilación. La tarea se ejecutará al llamar a su método <code>call</code> . Permite especificar la lista de procesadores de anotaciones con <code>setProcessors</code> .
Tool y OptionChecker	Interfaces para la ejecución de herramientas y comprobación de opciones.
ToolProvider	Clase factoría que devuelve una implementación de un JavaCompiler .
JavaFileManager	Interfaz del sistema de ficheros que proporciona ficheros al compilador.
StandardJavaFileManager	Interfaz del sistema de ficheros estándar que devuelve el JavaCompiler .
ForwardingJavaFileManager	Permite delegación de funciones entre llamadas de un JavaFileManager .
Diagnostic	Interfaz que modela los mensajes de diagnóstico emitidos por el compilador.
DiagnosticListener	Interfaz para recibir los mensajes de diagnóstico emitidos por el compilador.
DiagnosticCollector	Implementación sencilla de DiagnosticListener que recoge los mensajes de diagnóstico emitidos por el compilador en la lista accesible en <code>getDiagnostics</code> .



javax.tools – Ficheros y Ubicaciones

Interfaz / Clase	Descripción
FileObject	Interfaz abstracción de los ficheros que manejarán las herramientas de la API.
ForwardingFileObject	Permite delegación de funciones entre llamadas de un FileObject .
JavaFileObject	Interfaz abstracción de los ficheros Java (tanto de código fuente como de clase).
JavaFileObject.Kind	Enumerado de tipos de ficheros Java: SOURCE , CLASS , HTML , OTHER .
SimpleJavaFileObject	Implementación básica de JavaFileObject con muchos métodos útiles.
ForwardingJavaFileObject	Permite delegación de funciones entre llamadas de un JavaFileObject .
StandardLocation	Enumerado con las diferentes rutas estándar para la ubicación de ficheros Java. Es utilizado también por la interfaz Filer de la API de procesamiento JSR 269. Cada uno de sus valores representa las diferentes rutas fundamentales para el proceso de compilación y generación de ficheros.



14.- COMPILER TREE API.

14.1.- Introducción.

La **Compiler Tree API** es una API similar a la Mirror API del procesamiento J2SE 1.5 o a la Language Model API de JSR 269 en el hecho de que modela el código fuente Java y **proporciona una vista del código fuente en tiempo de compilación, de sólo lectura, pero para representar las construcciones sintácticas y sus relaciones jerárquicas**. Para más información, puede consultar la documentación oficial de la Compiler Tree API para Java SE 6 [<http://docs.oracle.com/javase/6/docs/jdk/api/javac/tree/>].

Algunos ejemplos de **usos típicos que se podrían dar a la Compiler Tree API** son:

- Facilitar la implementación de analizadores de código gracias a que proporciona una acceso más directo a las estructuras sintácticas del código fuente, así como a su texto.
- Facilitar, junto a [JSR 198](#) (Standard Extension API for IDEs), la implementación de IDEs, al permitir generar un meta-modelo del código Java, que podría servir de base sobre la que construir la implementación de los servicios avanzados habituales de un entorno de desarrollo.
- Acceso a información sintáctica del código fuente durante el procesamiento de anotaciones. La Compiler Tree API puede proporcionarnos información adicional de naturaleza más sintáctica en la implementación de procesadores para análisis y validación de código fuente.

IMPORTANTE: La Compiler Tree API está relacionada con la Java Compiler API (JSR 199) y la API de procesamiento JSR 269. Este es el principal motivo de dedicar un apartado del presente manual a examinar los componentes básicos de la misma. La relación entre dichas APIs se establece a través de la clase `com.sun.source.util.Trees`, en cuya documentación oficial se describe que “Puentea las APIs JSR 199, JSR 269 y la Tree API”. Como veremos, la clase de utilidad `Trees` puede extraer información sintáctica instanciándose a partir tanto de una instancia de la interfaz `JavaCompiler.CompilationTask` de la Java Compiler API, como de una instancia de `ProcessingEnvironment` de la API JSR 269. Y es que, como veremos, **Compiler Tree API nos permitirá descubrir enotaciones que no descubre la API JSR 269, como las anotaciones sobre variables locales y otras (!!).**

IMPORTANTE: Compiler Tree API no es una API estándar de la plataforma Java: sólo está oficialmente soportada por el compilador javac de Sun/Oracle, ya que no ha pasado a través del JCP (Java Community Process) y, por ello, su paquete raíz sigue haciendo referencia a Sun Microsystems (`com.sun.source`) y no a `java` o `javax`, como es el caso de las APIs estándar. Además, tampoco forma parte del entorno de ejecución (fichero `rt.jar`), si no que se concibe como parte de las herramientas de desarrollo, por lo que sus clases están incluidas en el fichero `tools.jar`, que únicamente existe en la JDK y no en la JRE.

Al igual que las APIs JSR 269 se han ido actualizando según se introducían novedades en el lenguaje Java, la Compiler Tree API también ha ido añadiendo soporte para las nuevas construcciones sintácticas de las últimas versiones de Java. En Java SE 8, además de añadir, entre otras, una `AnnotatedTypeTree`, se incluye el paquete `com.sun.source.doctree` para representar también los comentarios de documentación (`/**`) como árboles sintácticos. Para más información puede consultarse la documentación oficial de la Compiler Tree API para Java SE 8 [<http://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/>].

La **Compiler Tree API** ofrece facilidades de navegación por el código fuente desde un punto de vista sintáctico mediante estructuras jerárquicas llamadas **árboles sintácticos abstractos** (AST = Abstract Syntax Trees). Los AST representan el código fuente como un árbol de nodos, cada uno de los cuales representa a su vez otra construcción sintáctica del lenguaje.

Todas las diferentes interfaces que representan construcciones sintácticas se encuentran en el paquete **com.sun.source.tree** y heredan de la interfaz base **Tree**. Por ejemplo, para modelar una clase, tenemos la **ClassTree**, que es subinterfaz de **StatementTree** (la interfaz que modela las sentencias del lenguaje Java), siendo esta última ya subinterfaz directa de **Tree**. Para más detalles acerca de cómo está modelada la jerarquía de construcciones sintácticas del lenguaje, pueden examinarse los diagramas de clase incluidos en este subapartado.

Todo nodo perteneciente a un AST es una instancia de alguna de las subinterfaces de **Tree**. Dicho nodo, que podríamos llamar “nodo padre”, podrá tener una serie de “nodos hijos” que representarán los componentes significativos de la construcción sintáctica dada por dicho “nodo padre”. De esta forma, por ejemplo, podemos tener una clase Java representada por un nodo instancia de la interfaz **ClassTree**, que contenga métodos representados por instancias de **MethodTree**, que a su vez contengan en su cuerpo (modelado por la interfaz **BlockTree**) declaraciones de variables locales al método representadas por instancias de **VariableTree**. Las anotaciones, por supuesto, también están modeladas con la interfaz **AnnotationTree**.

14.2.- Paquete com.sun.source.tree.

A continuación, incluimos los diagrama de clases de **com.sun.source.tree**, donde puede observarse la jerarquía de construcciones sintácticas. Algunas subinterfaces heredan directamente de **Tree**, las que modelan sentencias heredan de **StatementTree** y las que modelan expresiones heredan de **ExpressionTree**.

Diagrama de clases: com.sun.source.tree (1/4): Subinterfaces directas de Tree

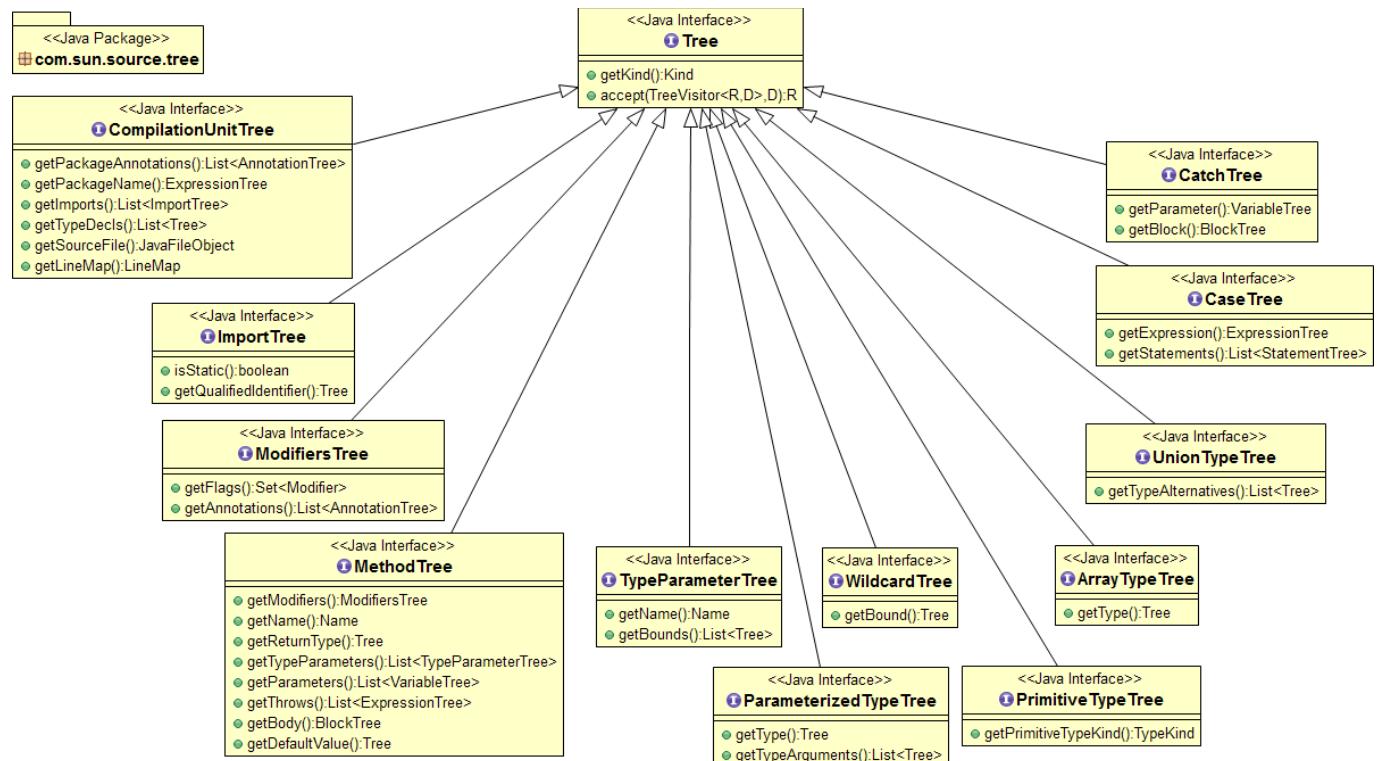


Diagrama de clases: com.sun.source.tree (2/4): Subinterfaces de StatementTree

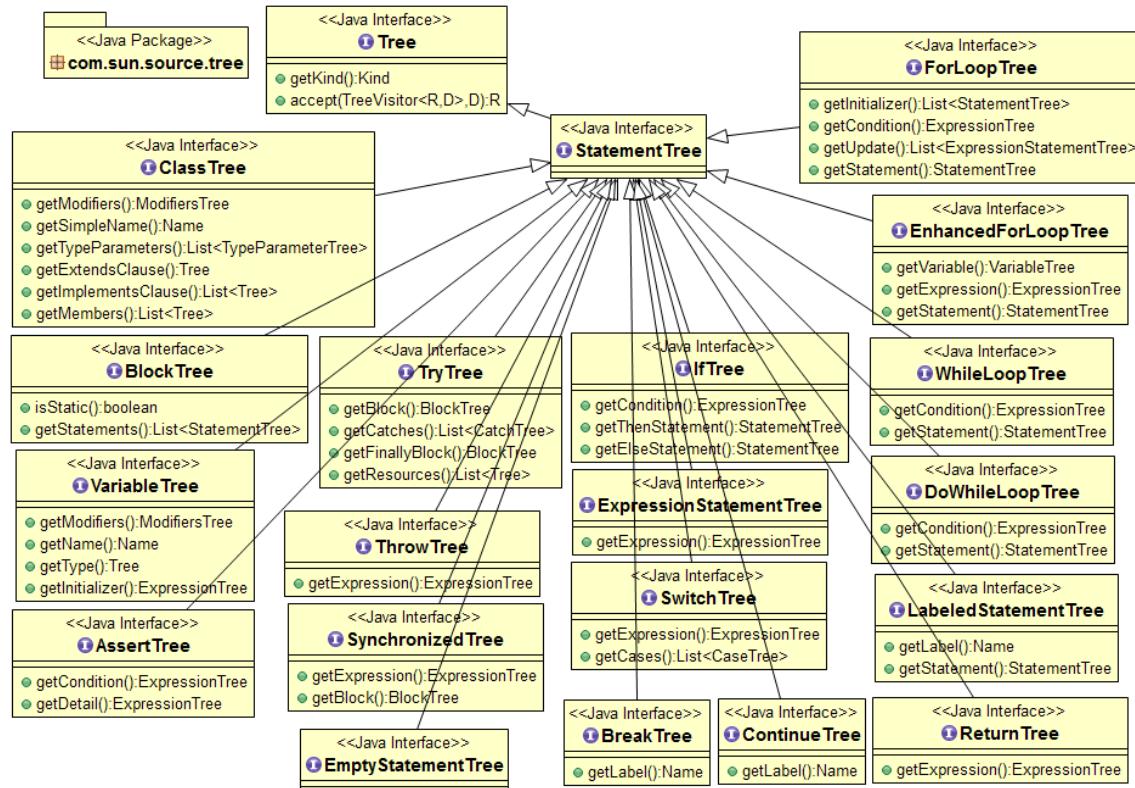


Diagrama de clases: com.sun.source.tree (3/4): Subinterfaces de ExpressionTree

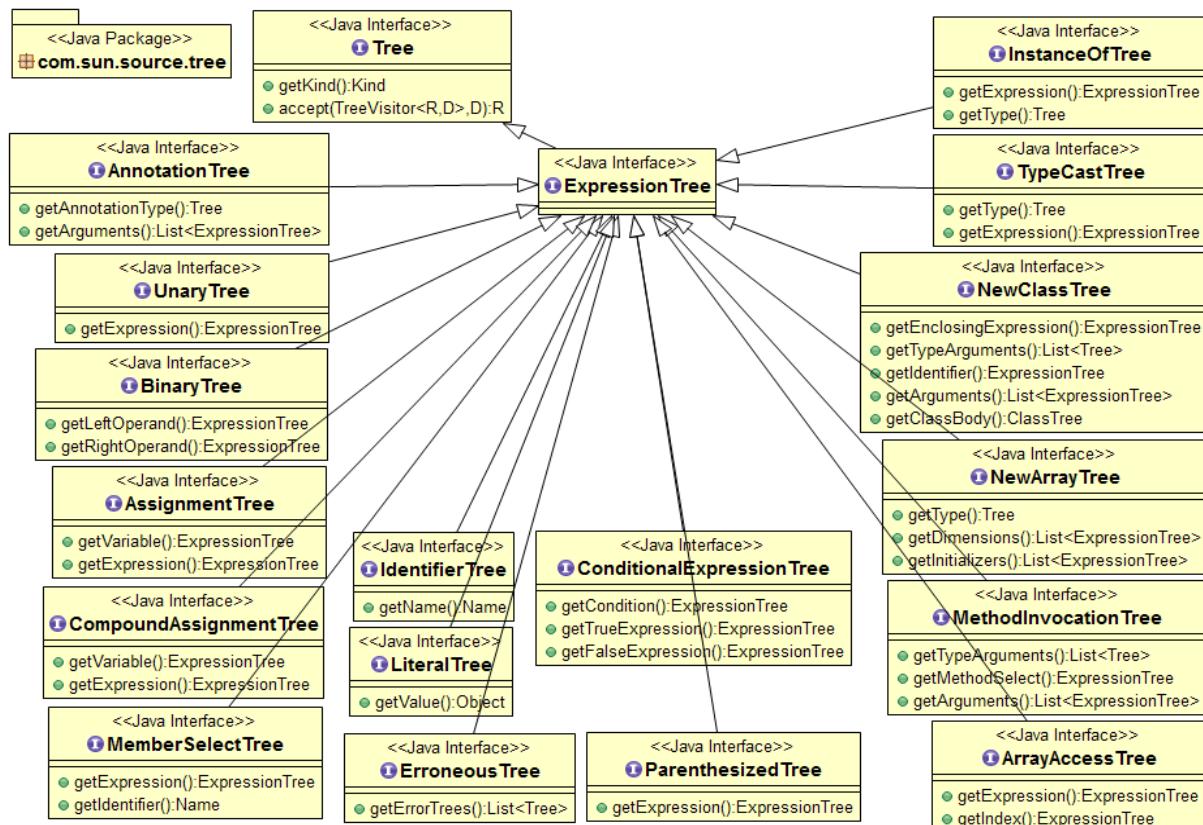


Diagrama de clases: com.sun.source.tree (4/4): TreeVisitor, Tree.Kind y otras

com.sun.source.tree – TreeVisitor, Tree.Kind y otras	
Interfaz / Clase	Descripción
Scope	Modela los ámbitos de programa y sirve para localizar grupos de elementos cercanos.
LineMap	Interfaz que permite convertir entre posiciones absolutas de carácter y números de línea.
TreeVisitor<R, P>	Visitor de nodos de los árboles sintácticos según el tipo de su construcción sintáctica.
Tree.Kind	Enumerado que modela la gran cantidad de tipos de construcciones sintácticas Java.

```

classDiagram
    package com.sun.source.tree {
        interface Scope
        interface LineMap
        interface TreeVisitor<R, P>
        enum Kind
    }
    class AnnotationTree, MethodInvocationTree, AssignmentTree, CompoundAssignmentTree, BinaryTree, BlockTree, BreakTree, CaseTree, CatchTree, ClassTree, ConditionalExpressionTree, ContinueTree, DoWhileLoopTree, EnhancedForLoopTree, ForLoopTree, IdentifierTree, ImportTree, ArrayAccessTree, LabeledStatementTree, LiteralTree, MethodTree, ModifiersTree, NewArrayTree, NewClassTree, ParenthesizedTree, ReturnTree, MemberSelectTree, EmptyStatementTree, SwitchTree, SynchronizedTree, ThrowTree, CompilationUnitTree, TryTree, ParameterizedTypeTree, UnionTypeTree, ArrayTypeTree, TypeCastTree, PrimitiveTypeTree, TypeParameterTree, InstanceOfTree, UnaryTree, VariableTree, WhileLoopTree, WildcardTree, OtherTree
    AnnotationTree <|-- MethodInvocationTree
    AnnotationTree <|-- AssignmentTree
    AnnotationTree <|-- CompoundAssignmentTree
    AnnotationTree <|-- BinaryTree
    AnnotationTree <|-- BlockTree
    AnnotationTree <|-- BreakTree
    AnnotationTree <|-- CaseTree
    AnnotationTree <|-- CatchTree
    AnnotationTree <|-- ClassTree
    AnnotationTree <|-- ConditionalExpressionTree
    AnnotationTree <|-- ContinueTree
    AnnotationTree <|-- DoWhileLoopTree
    AnnotationTree <|-- EnhancedForLoopTree
    AnnotationTree <|-- ForLoopTree
    AnnotationTree <|-- IdentifierTree
    AnnotationTree <|-- ImportTree
    AnnotationTree <|-- ArrayAccessTree
    AnnotationTree <|-- LabeledStatementTree
    AnnotationTree <|-- LiteralTree
    AnnotationTree <|-- MethodTree
    AnnotationTree <|-- ModifiersTree
    AnnotationTree <|-- NewArrayTree
    AnnotationTree <|-- NewClassTree
    AnnotationTree <|-- ParenthesizedTree
    AnnotationTree <|-- ReturnTree
    AnnotationTree <|-- MemberSelectTree
    AnnotationTree <|-- EmptyStatementTree
    AnnotationTree <|-- SwitchTree
    AnnotationTree <|-- SynchronizedTree
    AnnotationTree <|-- ThrowTree
    AnnotationTree <|-- CompilationUnitTree
    AnnotationTree <|-- TryTree
    AnnotationTree <|-- ParameterizedTypeTree
    AnnotationTree <|-- UnionTypeTree
    AnnotationTree <|-- ArrayTypeTree
    AnnotationTree <|-- TypeCastTree
    AnnotationTree <|-- PrimitiveTypeTree
    AnnotationTree <|-- TypeParameterTree
    AnnotationTree <|-- InstanceOfTree
    AnnotationTree <|-- UnaryTree
    AnnotationTree <|-- VariableTree
    AnnotationTree <|-- WhileLoopTree
    AnnotationTree <|-- WildcardTree
    AnnotationTree <|-- OtherTree
    Kind {
        MULTPLY: Kind
        DIVIDE: Kind
        REMAINDER: Kind
        PLUS: Kind
        MINUS: Kind
        LEFT_SHIFT: Kind
        RIGHT_SHIFT: Kind
        UNSIGNED_RIGHT_SHIFT: Kind
        LESS_THAN: Kind
        GREATER_THAN: Kind
        LESS_THAN_EQUAL: Kind
        GREATER_THAN_EQUAL: Kind
        EQUAL_TO: Kind
        NOT_EQUAL_TO: Kind
        AND: Kind
        XOR: Kind
        OR: Kind
        CONDITIONAL_AND: Kind
        CONDITIONAL_OR: Kind
        MULTIPLY_ASSIGNMENT: Kind
        DIVIDE_ASSIGNMENT: Kind
        REMAINDER_ASSIGNMENT: Kind
        PLUS_ASSIGNMENT: Kind
        MINUS_ASSIGNMENT: Kind
        LEFT_SHIFT_ASSIGNMENT: Kind
        RIGHT_SHIFT_ASSIGNMENT: Kind
        UNSIGNED_RIGHT_SHIFT_ASSIGNMENT: Kind
        AND_ASSIGNMENT: Kind
        XOR_ASSIGNMENT: Kind
        OR_ASSIGNMENT: Kind
        INT_LITERAL: Kind
        LONG_LITERAL: Kind
        FLOAT_LITERAL: Kind
        DOUBLE_LITERAL: Kind
        BOOLEAN_LITERAL: Kind
        CHAR_LITERAL: Kind
        STRING_LITERAL: Kind
        NULL_LITERAL: Kind
        UNBOUNDED_WILDCARD: Kind
        EXTENDS_WILDCARD: Kind
        SUPER_WILDCARD: Kind
        ERRONEOUS: Kind
        INTERFACE: Kind
        ENUM: Kind
        ANNOTATION_TYPE: Kind
        OTHER: Kind
        associatedInterface: Class<Tree>
        VALUES: Kind[]
        values(): Kind[]
        valueOf(String): Kind
        Kind(Class<Tree>)
        asInterface(): Class<Tree>
        <clinit>(): void
    }
    Scope <|-- LineMap
    LineMap <|-- TreeVisitor<R, P>
    TreeVisitor<R, P> <|-- AnnotationTree
    TreeVisitor<R, P> <|-- MethodInvocationTree
    TreeVisitor<R, P> <|-- AssignmentTree
    TreeVisitor<R, P> <|-- CompoundAssignmentTree
    TreeVisitor<R, P> <|-- BinaryTree
    TreeVisitor<R, P> <|-- BlockTree
    TreeVisitor<R, P> <|-- BreakTree
    TreeVisitor<R, P> <|-- CaseTree
    TreeVisitor<R, P> <|-- CatchTree
    TreeVisitor<R, P> <|-- ClassTree
    TreeVisitor<R, P> <|-- ConditionalExpressionTree
    TreeVisitor<R, P> <|-- ContinueTree
    TreeVisitor<R, P> <|-- DoWhileLoopTree
    TreeVisitor<R, P> <|-- EnhancedForLoopTree
    TreeVisitor<R, P> <|-- ForLoopTree
    TreeVisitor<R, P> <|-- IdentifierTree
    TreeVisitor<R, P> <|-- ImportTree
    TreeVisitor<R, P> <|-- ArrayAccessTree
    TreeVisitor<R, P> <|-- LabeledStatementTree
    TreeVisitor<R, P> <|-- LiteralTree
    TreeVisitor<R, P> <|-- MethodTree
    TreeVisitor<R, P> <|-- ModifiersTree
    TreeVisitor<R, P> <|-- NewArrayTree
    TreeVisitor<R, P> <|-- NewClassTree
    TreeVisitor<R, P> <|-- ParenthesizedTree
    TreeVisitor<R, P> <|-- ReturnTree
    TreeVisitor<R, P> <|-- MemberSelectTree
    TreeVisitor<R, P> <|-- EmptyStatementTree
    TreeVisitor<R, P> <|-- SwitchTree
    TreeVisitor<R, P> <|-- SynchronizedTree
    TreeVisitor<R, P> <|-- ThrowTree
    TreeVisitor<R, P> <|-- CompilationUnitTree
    TreeVisitor<R, P> <|-- TryTree
    TreeVisitor<R, P> <|-- ParameterizedTypeTree
    TreeVisitor<R, P> <|-- UnionTypeTree
    TreeVisitor<R, P> <|-- ArrayTypeTree
    TreeVisitor<R, P> <|-- TypeCastTree
    TreeVisitor<R, P> <|-- PrimitiveTypeTree
    TreeVisitor<R, P> <|-- TypeParameterTree
    TreeVisitor<R, P> <|-- InstanceOfTree
    TreeVisitor<R, P> <|-- UnaryTree
    TreeVisitor<R, P> <|-- VariableTree
    TreeVisitor<R, P> <|-- WhileLoopTree
    TreeVisitor<R, P> <|-- WildcardTree
    TreeVisitor<R, P> <|-- OtherTree
    Kind <|-- MULTPLY: Kind
    Kind <|-- DIVIDE: Kind
    Kind <|-- REMAINDER: Kind
    Kind <|-- PLUS: Kind
    Kind <|-- MINUS: Kind
    Kind <|-- LEFT_SHIFT: Kind
    Kind <|-- RIGHT_SHIFT: Kind
    Kind <|-- UNSIGNED_RIGHT_SHIFT: Kind
    Kind <|-- LESS_THAN: Kind
    Kind <|-- GREATER_THAN: Kind
    Kind <|-- LESS_THAN_EQUAL: Kind
    Kind <|-- GREATER_THAN_EQUAL: Kind
    Kind <|-- EQUAL_TO: Kind
    Kind <|-- NOT_EQUAL_TO: Kind
    Kind <|-- AND: Kind
    Kind <|-- XOR: Kind
    Kind <|-- OR: Kind
    Kind <|-- CONDITIONAL_AND: Kind
    Kind <|-- CONDITIONAL_OR: Kind
    Kind <|-- MULTIPLY_ASSIGNMENT: Kind
    Kind <|-- DIVIDE_ASSIGNMENT: Kind
    Kind <|-- REMAINDER_ASSIGNMENT: Kind
    Kind <|-- PLUS_ASSIGNMENT: Kind
    Kind <|-- MINUS_ASSIGNMENT: Kind
    Kind <|-- LEFT_SHIFT_ASSIGNMENT: Kind
    Kind <|-- RIGHT_SHIFT_ASSIGNMENT: Kind
    Kind <|-- UNSIGNED_RIGHT_SHIFT_ASSIGNMENT: Kind
    Kind <|-- AND_ASSIGNMENT: Kind
    Kind <|-- XOR_ASSIGNMENT: Kind
    Kind <|-- OR_ASSIGNMENT: Kind
    Kind <|-- INT_LITERAL: Kind
    Kind <|-- LONG_LITERAL: Kind
    Kind <|-- FLOAT_LITERAL: Kind
    Kind <|-- DOUBLE_LITERAL: Kind
    Kind <|-- BOOLEAN_LITERAL: Kind
    Kind <|-- CHAR_LITERAL: Kind
    Kind <|-- STRING_LITERAL: Kind
    Kind <|-- NULL_LITERAL: Kind
    Kind <|-- UNBOUNDED_WILDCARD: Kind
    Kind <|-- EXTENDS_WILDCARD: Kind
    Kind <|-- SUPER_WILDCARD: Kind
    Kind <|-- ERRONEOUS: Kind
    Kind <|-- INTERFACE: Kind
    Kind <|-- ENUM: Kind
    Kind <|-- ANNOTATION_TYPE: Kind
    Kind <|-- OTHER: Kind
    Kind <|-- associatedInterface: Class<Tree>
    Kind <|-- VALUES: Kind[]
    Kind <|-- values(): Kind[]
    Kind <|-- valueOf(String): Kind
    Kind <|-- Kind(Class<Tree>)
    Kind <|-- asInterface(): Class<Tree>
    Kind <|-- <clinit>(): void
  
```

Como se ve, la Compiler Tree API, al igual que las otras dos APIs que modelan el lenguaje Java, la Mirror API y la Model API, también tiene una interfaz Visitor, **TreeVisitor<R, P>**, para poder navegar visitando las jerarquías de nodos de los árboles sintácticos. Eso sí, dado el enorme número de construcciones sintácticas diferentes que existen en el lenguaje Java, la interfaz **TreeVisitor<R, P>** define un número de métodos verdaderamente escalofriante.

NOTA: El tipo enumerado **Tree.Kind**, que tiene un valor para representar todas y cada una de las construcciones sintácticas del lenguaje Java según se representan en los árboles sintácticos de la Compiler Tree API, tiene tal cantidad de valores diferentes que ha tenido que partirse en dos su representación para que pudiera ajustarse correctamente al tamaño de la página.

14.3.- Paquete com.sun.source.util.

com.sun.source.util – Implementaciones de Visitors y Tareas de parseo	
Interfaz / Clase	Descripción
Trees	Utilidades para árboles sintácticos. Se construye instancias de CompilationTask (de la Java Compiler API) o de ProcessingEnvironment (de la API JSR 269).
TreePath	Ruta que contiene el camino desde la raíz de la unidad de compilación hasta un nodo.
JavacTask	Acceso a las facilidades del compilador para parsear, analizar e incluso generar código.
TaskListener/Event	Listener y evento para notificación de eventos generados por las tareas del compilador.
SimpleTreeVisitor	Implementación sencilla de TreeVisitor . Todos los métodos llaman a defaultAction .
Tree[Path]Scanner	Visitor de nodos y los subnodos. treePathScanner permite obtener el TreePath actual.
SourcePositions	Obtención de posiciones de una instancia de Tree dentro de una unidad de compilación.



15.- PROCESAMIENTO JSR 269 (JAVA SE 6+).

15.1.- Introducción.

La especificación [JSR 269](#) (Pluggable Annotation Processing API) fue introducida como parte de Java SE 6 para estandarizar y mejorar las funcionalidades y usabilidad del procesamiento de anotaciones en base a toda la experiencia recogida con **apt**.

apt se había revelado como una herramienta no estándar (sólo se incluía en la JDK de Sun Microsystems), que además era incómoda de usar y tenía un gran número de fallos y limitaciones, de los que hemos hablado en el apartado “**Fallos conocidos y limitaciones de apt**”.

Siendo así, **apt** y la **Mirror API** fueron deprecados en Java SE 6 y se introdujeron las nuevas API estándar destinadas a reemplazarlas:

- `javax.lang.model.(element/type/util)` en vez de `com.sun.mirror.(declaration/type/util)`
- `javax.annotation.processing` en vez de `com.sun.mirror.apt`.
- `javac` en lugar de **apt** como herramienta de procesamiento de anotaciones.

Por tanto, podemos decir que los **componentes fundamentales del procesamiento de anotaciones JSR 269** son los siguientes:

- API del procesamiento como tal para procesar código (`javax.annotation.processing`).
- API modelo de Java para representar código en tiempo de compilación (`javax.lang.model`).
- Herramienta para la ejecución del procesamiento de anotaciones (`javac`).

El procesamiento de anotaciones JSR 269 hace trabajar juntos todos estos componentes para implementar nuevas funcionalidades, como análisis de código, validación y generación de nuevos ficheros, en base a la meta-information añadida por las anotaciones a los elementos de código anotados, según el siguiente esquema:

- (1) **ficheros .java**: ficheros de código Java anotado con ciertos tipos anotación.
- (2) **procesadores de anotaciones**: procesarán dichos tipos anotación sobre código anotado.
- (3) **javac**: herramienta de procesamiento que aplicará los procesadores al código anotado.

El procesamiento se basa en una **estrategia parecida a los preprocesadores** de otros lenguajes, donde **los procesadores de anotaciones ejercen el rol clásico de un preprocesador**, procesando el código inicial antes de ser enviado finalmente al compilador. Sin embargo, **a diferencia de un preprocesador tradicional, el “preprocesamiento JSR 269” no es fijo, si no que viene dado por la lógica implementada de forma individual por cada procesador de anotaciones**.

Para implementar la lógica de los procesadores JSR 269, se hacen necesarias tanto una API propia para el procesamiento (`javax.annotation.processing`), como una API que modele los elementos del lenguaje Java sobre los que se va a trabajar (`javax.lang.model`). Además de una serie de consideraciones únicas, como que, al existir la posibilidad de que pueda generarse nuevo código fuente durante el preprocesamiento, se necesiten múltiples iteraciones de preprocesamiento (las denominadas “rondas”). En los siguientes subapartados se explican en detalle todos los aspectos del procesamiento de anotaciones JSR 269.

15.2.- Cómo funciona el procesamiento JSR 269.

Invocación del procesamiento de anotaciones

El procesamiento de anotaciones JSR 269 comienza con una invocación a **javac**, el compilador Java. Para invocar el procesamiento de anotaciones no hay que especificar ninguna opción en particular: **javac por defecto ya realiza el procesamiento de anotaciones**.

Asimismo, javac proporciona una serie de opciones de línea de comandos específicas para procesamiento de anotaciones con las es posible configurar todos los aspectos del proceso.

Típicamente se suelen indicar las opciones habituales de compilación de **javac** más algunas opciones adicionales específicas del procesamiento de anotaciones, como el procesador o procesadores de anotaciones a utilizar o las rutas donde buscar procesadores de anotaciones (en el llamado proceso de “descubrimiento” / “discovery”), el directorio de salida para los ficheros generados durante el procesamiento, los ficheros de código fuente a procesar, etcétera.

Sin entrar en detalles (para lo cual puede consultarse el subapartado “Opciones de línea de comando de **javac**”), supongamos que iniciamos el procesamiento de anotaciones con una línea de comando de **javac** similar a la siguiente:

```
javac -cp CLASSPATH .\src\paquete\ClaseAnotada.java
```

En la línea de comandos se indica el classpath para la compilación, así como los **ficheros de código fuente sobre los que se desea realizar el procesamiento de anotaciones**.

Ronda 0 de procesamiento

Nada más invocarse el procesamiento de anotaciones, **javac** entra en la llamada “ronda 0” de procesamiento (un concepto que no existía en **apt**, donde se empezaba directamente en la “ronda 1” o “primera ronda” de procesamiento).

En la ronda 0 sólo se llevará a cabo el descubrimiento de los procesadores de anotaciones disponibles dentro de la ruta de búsqueda para la que se haya determinado que hay que buscar.

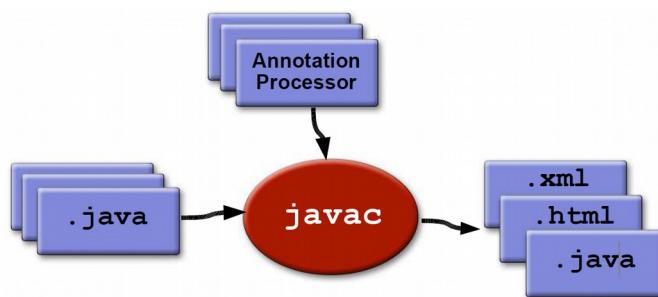
Descubrimiento de procesadores de anotaciones

javac lleva a cabo el denominado el proceso de descubrimiento (en inglés: *discovery*) de los procesadores de anotación disponibles. Hay varias reglas que rigen este proceso, pero básicamente se trata de buscar ficheros **META-INF/services/javax.annotation.processing.Processor**, que contienen los nombres canónicos de las clases de los procesadores, recorriendo toda la “ruta de búsqueda” que se determine. Para más detalles sobre este proceso, puede consultarse el subapartado “Funcionamiento de **javac**”.



Ronda 1 de procesamiento de anotaciones

Una vez que se ha realizado el descubrimiento de los procesadores de anotaciones disponibles, se inicia la “ronda 1” o “primera ronda” de procesamiento. Como entrada a esta primera ronda se toman los ficheros especificados en la invocación inicial al procesamiento.



Sobre los ficheros que se pasan como entrada a **javac** se hará:

- 1) Un análisis para la **búsqueda de tipos anotación a procesar** que contienen.
- 2) Un **emparejamiento de procesadores de anotaciones ↔ tipos anotación**.
- 2 & 3) El **procesamiento de anotaciones** como tal, mientras se realiza el emparejamiento.

Búsqueda de tipos anotación a procesar

Al inicio de cada ronda de procesamiento, **javac** realiza una **búsqueda de tipos anotación** presentes en el código fuente que se va a procesar para saber los tipos anotación que tendrá que procesar mediante procesamiento de anotaciones. En este caso, se analizará el texto del código fuente del fichero **ClaseAnotada.java** para encontrar qué anotaciones contiene. Supongamos que **javac** encuentra varias anotaciones del tipo anotación ficticio **@AnotacionEjemplo**.



Emparejamiento de procesadores de anotaciones ↔ tipos anotación

Una vez que **javac** ha determinado qué tipos anotación tiene que procesar, conociendo de la “ronda 0” con qué procesadores cuenta, realiza un emparejamiento entre procesadores y tipos anotación a procesar. Para ello, sigue el siguiente **procedimiento de emparejamiento**:

- 1) **javac** solicita a cada procesador una lista de los tipos anotación que procesa.
- 2) **javac** asignará el tipo anotación al procesador si procesa dicho tipo anotación.
- 3) **javac** invoca al procesador asignado y, si este “reclama” (del inglés: “claim”) dicho tipo anotación, el tipo saldrá de la lista de tipos anotación pendientes. Si el procesador no “reclama” el tipo anotación para sí mismo, **javac** buscará más procesadores aplicables al tipo anotación.
- 4) Este proceso continuará hasta que todos los tipos anotación hayan sido “reclamados” (y, por tanto, la lista de tipos anotación pendientes es vacía) o no haya más procesadores aplicables.

Interacción de javac con los procesadores de anotaciones

Cuando **javac** necesita interactuar con un procesador de anotaciones para interrogarlo, por ejemplo, sobre los tipos anotación que soporta, para poder llevar a cabo el procedimiento de emparejamiento, sigue los siguientes pasos:

- 1) Si no ha sido instanciado con anterioridad, **javac** creará una instancia de dicho procesador llamando al constructor sin argumentos de la clase que implementa el procesador.
- 2) **javac** llama al método **init** del procesador pasándole un **ProcessingEnvironment**.
- 3) **javac** llama a todos los métodos que le permiten recabar información acerca del procesador: **getSupportedAnnotationTypes**, **getSupportedOptions**, y **getSupportedSourceVersion**.

Como vemos, mediante la exigencia de que las clases procesador tengan un constructor sin argumentos, el método **init** y los demás métodos de información para el proceso de emparejamiento, se hace innecesaria la existencia de las factorías del procesamiento J2SE 1.5.

Procesamiento de anotaciones

Una vez que **javac** logra emparejar un procesador con un tipo anotación, para que este lleve a cabo el procesamiento correspondiente, llama al método **process** del procesador:

```
boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)
```

El método **process** recibe como parámetros un conjunto de tipos anotación que son los que el procesador deberá procesar (**CUIDADO**: los tipos anotación, no las anotaciones en sí) y una instancia de **RoundEnvironment**, el “entorno de la ronda”, que nos proporcionará toda la información necesaria para implementar el procesamiento en la ronda actual.

Es muy importante comprender el **valor de retorno boolean del método process** porque influirá en el resto de procesadores que podrían invocarse sobre los mismos tipos anotación. En el apartado de la página anterior en el que describíamos el emparejamiento entre procesadores y tipos anotación, se decía que un procesador podía “reclamar” o no un tipo o tipos anotación. Pues bien, esto es precisamente lo que indica el tipo de retorno **boolean** del método **process**, si este procesador “reclama” para sí mismo los tipos de anotación pasados como argumento. En caso de que los reclame, estos tipos anotación salen de la lista de tipos anotación pendientes y ya no podrán ser reclamados por otro procesador de anotaciones dentro de la presente ronda. Normalmente, dado que los procesadores de anotaciones no deberían interferir unos con otros, se recomienda abiertamente devolver **false** para no impedir la ejecución de otros procesadores de anotaciones, a no ser que se tenga una buena razón para ello.

Lo anterior es especialmente delicado en el caso de los “procesadores universales”, que son procesadores especiales que pueden invocarse incluso aunque el código fuente no tenga ni un sólo elemento anotado. Este tipo de procesadores están encaminados a tareas de propósito general sobre todas los ficheros de código fuente sin excepción. Su tipo anotación soportado es “*” que significa “todos los tipos anotación”. Cuando **javac** llama al método **process** de este tipo de “procesadores universales” el conjunto de tipos anotación se pasa vacio. En ese caso, además, si el procesador devolviera **true**, reclamaría todos los tipos anotación para sí mismo y evitaría que se ejecutara cualquier otro procesador de anotaciones. En este caso concreto, incluso en la documentación oficial se recomienda que se devuelva **false** para evitar impedir que se ejecuten otros procesadores de anotaciones, ya fueran universales o no.

El procesamiento de anotaciones dentro de una ronda continuará mientras queden tipos anotación pendientes que no hayan sido emparejados. Si todos los procesadores fueran “reclamando” sus tipos anotaciones, la lista llegaría a vaciarse y ya no se seguirían buscando más procesadores de anotaciones. Si no, el procesamiento terminará, aunque queden tipos anotación pendientes de procesar, cuando se ejecuten todos los procesadores disponibles.

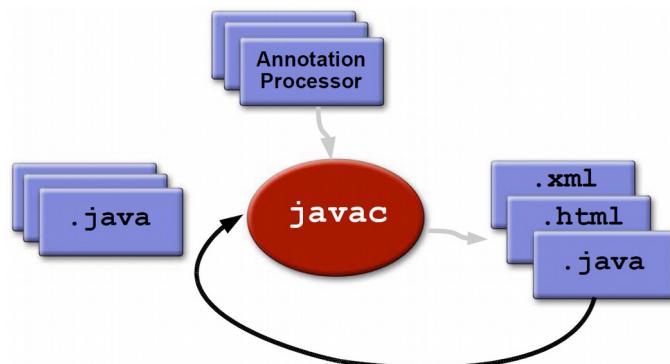
Cuando todos y cada uno de los procesadores de anotaciones descubiertos y emparejados hayan finalizado la ejecución de su método **process**, terminará la “ronda 1” o “primera ronda” del procesamiento de anotaciones. Y, si cualquier procesador ha generado nuevos ficheros de código fuente durante la ronda, incluso se producirá una “ronda adicional”.

Rondas adicionales de procesamiento de anotaciones (si procede)

El concepto de “ronda” (del inglés: “round”) surge debido al hecho de que los procesadores de anotaciones pueden generar nuevo código fuente. Y dicho nuevo código fuente generado ¡¡ipodría a su vez contener anotaciones que es necesario procesar!!!

Es decir, que **mientras que haya procesadores de código que generen nuevo código fuente, el procesamiento de anotaciones no puede finalizar, debe continuar con una nueva “ronda adicional de procesamiento”.**

Si se hace necesaria una ronda adicional de procesamiento debido a que algún procesador de anotaciones ha generado nuevo código fuente, **javac** pasará a realizar de nuevo los mismos pasos que se llevaron a cabo durante la primera ronda, pero esta vez lo que se tomará como entrada de código fuente a procesar será el nuevo código fuente recién generado en la ronda anterior, como se ilustra en el esquema:



javac realizará de nuevo los mismos procesos que en la primera ronda:

- 1) Un análisis para la búsqueda de tipos anotación a procesar que contienen.
- 2) Un emparejamiento de procesadores de anotaciones ↔ tipos anotación.
- 2 & 3) El procesamiento de anotaciones como tal, mientras se realiza el emparejamiento.

IMPORTANTE: Como hemos comentado cuando nos referímos a la primera ronda, en las rondas subsiguientes no se instanciarán de nuevo los procesadores que ya hubieran sido instanciados con anterioridad. **Crear una instancia, llamar a `init` y a los demás métodos informativos sólo se hace una vez por procesador** dentro de una misma ejecución del procesamiento. Además, ocurre otro comportamiento bastante particular: **en cada ronda adicional siempre se ejecutarán de nuevo todos los procesadores ejecutados con anterioridad, aunque no tengan anotaciones que procesar.**

Ronda final del procesamiento de anotaciones

Una vez se haya completado una ronda de procesamiento sin haber generado nuevo código fuente, se entra en la llamada “ronda final”.

En la “ronda final”, `javac` invocará por última vez a todos los procesadores de anotaciones que se hayan ejecutado en cualquiera de las rondas anteriores. Esto es así para ofrecer una oportunidad a los procesadores de anotaciones JSR 269 de cerrar tareas o recursos que pudieran haber querido tener abiertos mientras que pudieran ocurrir nuevas rondas de procesamiento.

Y es que algunos procesadores podrían estar diseñados para trabajar en múltiples rondas y, por cuestiones de eficiencia, no obtener y liberar sus recursos en cada ronda, si no de esta forma inicializarlos al principio y liberarlos cuando se encuentran en la “ronda final”.

Compilación de los resultados del proc. anotaciones (si procede)

Una vez ejecutados todos los procesadores en la “ronda final” se da por terminado el procesamiento y, por defecto, `javac` realizará la compilación de todos los ficheros de código fuente, tanto los que había originalmente antes del procesamiento, como todos los nuevos que se hayan podido generar a lo largo de todas las rondas de procesamiento de anotaciones.

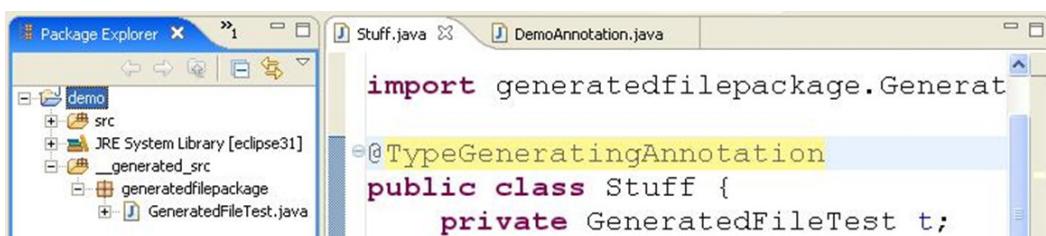
NOTA: La compilación final de `javac` puede no darse si se especifica la opción `-proc:only` en la línea de comandos. Esta opción, como veremos más adelante, puede tener mucho interés a la hora de depurar errores cometidos en los ficheros generados por los procesadores.

Resultados del procesamiento de anotaciones

Los resultados generados por el procesamiento de anotaciones pueden ser, como ya sabemos, de varios tipos, pero pueden agruparse en dos grandes categorías:

- (1) **Nuevos ficheros generados**, ya sea de código fuente u otro tipo de ficheros.
- (2) **Mensajes de compilación**, ya sean mensajes informativos, warnings o errores.

Los ficheros normalmente se generarán en el directorio configurado en `javac` como directorio de salida de nuevos ficheros usando su opción `-s DIRECTORIO`. Asimismo, si se usa procesamiento de anotaciones en un IDE, como [Eclipse](#) o [NetBeans](#), aparecerán reflejados los nuevos ficheros en la vista correspondiente. La siguiente imagen muestra un nuevo fichero generado en el directorio configurado para ello dentro de un proyecto de [Eclipse](#):



Los mensajes arrojados por los procesadores de anotaciones durante el procesamiento podrán leerse directamente en la consola de comandos si se usa `javac` por línea de comandos, o en el caso de usar un IDE, igualmente en la vista que oportunamente establezca dicho IDE.

15.3.- Componentes del procesamiento JSR 269.

Como ya mencionamos en la introducción, el procesamiento de anotaciones consta de varias entidades. Los **principales componentes del procesamiento JSR 269** son:

- API de procesamiento de anotaciones (paquete `javax.annotation.processing`).
- API modelo de Java (Model API) en tiempo de compilación (paquetes `javax.lang.model`).
- Herramienta para la ejecución del procesamiento de anotaciones (el compilador `javac`).

Con la API de procesamiento (`javax.annotation.processing`) escribiremos el código de los procesadores de anotaciones, en los que procesaremos los elementos del lenguaje representados por las interfaces de la Model API (paquetes `javax.lang.model`). Finalmente, ejecutaremos el procesamiento de anotaciones como uno de los procesos iniciales de `javac`.

API de procesamiento: Soporte de herramientas de desarrollo (“tooling”)

Los diseñadores responsables de `apt` y el procesamiento J2SE 1.5 aprendieron la lección de lo fundamental que era la integración del procesamiento con las herramientas de desarrollo más habituales. Empezaron por lo más importante y fastidioso de `apt`: el hecho de que fuera una herramienta externa, por lo que, en JSR 269, ya es el compilador directamente el que integra el procesamiento de anotaciones. Y es que a la hora de elaborar la especificación JSR 269 se han tenido más en cuenta los entornos y las herramientas de desarrollo.

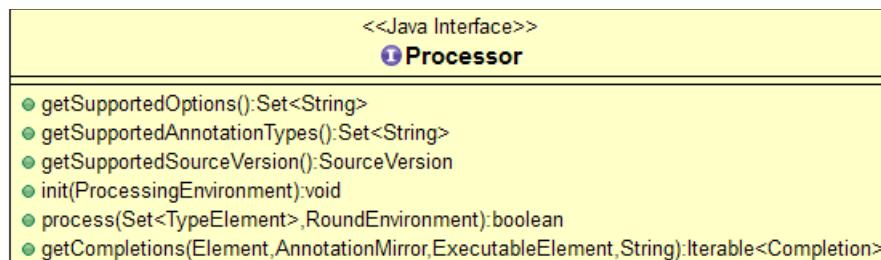
El diseño de la nueva API de procesamiento tiene en cuenta la posibilidad de ejecutarse en un entorno de compilación incremental. Así, de hecho, ocurre en la implementación del procesamiento de anotaciones del compilador de [Eclipse](#), que, a diferencia de `javac`, sí soporta compilación incremental. Por ejemplo, en la interfaz `Filer`, los métodos de creación de ficheros son métodos de argumentos variables (varargs) que reciben como argumentos para la creación de ficheros la lista de elementos que originan el fichero, sugiriéndose en la documentación que esta información podría utilizarse en un entorno de compilación incremental para determinar la necesidad de volver a ejecutar los procesadores y/o borrar los ficheros generados.

Además, como ya veremos, los procesadores de anotaciones JSR 269 han incorporado un método `getCompletions` que devuelve sugerencias de “Completions” o “Terminaciones” de código. Este es un mecanismo de comunicación pensado expresamente para que los IDEs puedan solicitar esta información a los procesadores a medida que sus usuarios van escribiendo código y ofrecer sugerencias para completar el código directamente.

15.3.1.- Procesadores de anotaciones JSR 269.

15.3.1.1.- Procesadores de anotaciones.

Los procesadores de anotaciones JSR 269 son simplemente clases que implementan la interfaz `javax.annotation.processing.Processor`.



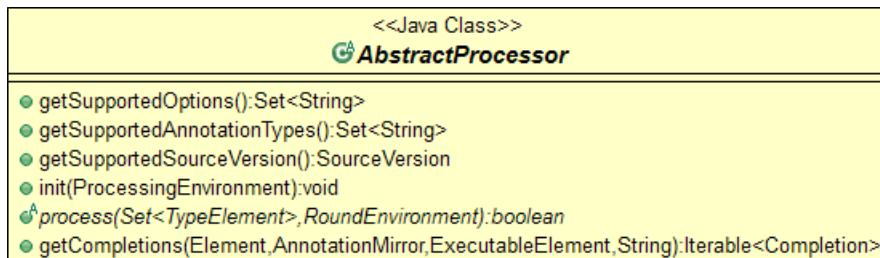
La interfaz `Processor` define todos los métodos fundamentales para la implementación del procesamiento de anotaciones y todas las fases que ello conlleva, incluido el descubrimiento de procesadores o el proceso de emparejamiento de procesadores con sus tipos anotación.

Métodos de la interfaz <code>javax.annotation.processing.Processor</code>	
Método	Descripción
<code>getSupportedAnnotationTypes</code>	Devuelve un conjunto de <code>Strings</code> con nombres canónicos de clases que define el conjunto de tipos anotación soportados por el procesador. Esto se usa en el emparejamiento de procesadores ↔ tipos anotación. Se puede usar comodines “*” parciales, como en “anotacion.procesamiento.*” o directamente “*” para TODOS los tipos anotación (“procesador universal”).
<code>getSupportedOptions</code>	Devuelve un conjunto de <code>Strings</code> con nombres de identificadores de las opciones soportadas por el procesador. En caso de que sea un nombre de varias palabras se sugiere utilizar el punto “.” como separador.
<code>getSupportedSourceVersion</code>	Devuelve una instancia del tipo enumerado <code>SourceVersion</code> para indicar cual es la última versión del nivel de código fuente Java soportada oficialmente por el procesador. Normalmente aquí se colocará la versión de la plataforma Java en que se hayan desarrollado los procesadores. Si hubiera sido en Java SE 7, devolveríamos <code>SourceVersion.RELEASE_7</code> .
<code>init</code>	Método de inicialización. En este método los procesadores de anotaciones deberán inicializar los campos y variables de procesamiento que sean necesarios para efectuar la lógica de procesamiento. Además, se pasa el objeto de entorno de procesamiento <code>ProcessingEnvironment</code> , vital ya que nos proporciona objetos de implementación de todas las interfaces de facilidades de procesamiento como <code>Messager</code> , <code>Filer</code> , <code>Elements</code> , etc.
<code>process</code>	Método de procesamiento. En el cuerpo de este método es donde la clase procesador realiza el procesamiento de anotaciones. Recibe el conjunto de tipos anotación a procesar (los tipos, no confundir con los elementos de las anotaciones a procesar) y un <code>RoundEnvironment</code> , el entorno de la ronda de procesamiento, que da información sobre la ronda actual y la anterior, además de ofrecer los métodos para el descubrimiento de los elementos a procesar: <code>getRootElement</code> s y <code>getElementsAnnotatedWith</code> . Como valor de retorno devuelve un <code>boolean</code> que indica si el procesador “reclama” los tipos anotación procesados en exclusiva. Si se devuelve <code>true</code> , los tipos anotación reclamados no serán procesados ya por ningún otro procesador. Por este motivo, se recomienda devolver <code>false</code> .
<code>getCompletions</code>	Devuelve una lista de sugerencias de “Completions” o “Terminaciones” de código en base a la información parcial que se le pasa como parámetro. Está pensado para ser llamado por los IDEs en tiempo real mientras sus usuarios escriben código para así recibir sugerencias de terminación de código para ahorrarse escribir código si seleccionan cualquiera de ellas.

Implementación de un proc. anotaciones JSR 269: `AbstractProcessor`

Para implementar un procesador de anotaciones propio, sería muy fácil implementar directamente la interfaz `Processor`, ya que no son muchos los métodos que define ni muy complicados de implementar, a excepción del método `process` propiamente dicho.

No obstante, la práctica más habitual e incluso recomendada por los diseñadores de la API a la hora de implementar un procesador de anotaciones no es directamente implementar la interfaz `Processor`, si no heredar de la clase abstracta `AbstractProcessor`, anotar la clase con `@SupportedAnnotationTypes`, `@SupportedOptions` y `@SupportedSourceVersion`, y luego redefinir los métodos `init` (si procede), `getCompletions` (si procede) y `process`.



La clase abstracta `AbstractProcessor` implementa la interfaz `Processor` y está pensada precisamente para hacer de clase base raíz, facilitando así la implementación de nuevos procesadores de anotaciones.

La siguiente tabla indica la implementación por defecto de cada uno de sus métodos, ya que será importante conocer las pequeñas particularidades de `AbstractProcessor` si, como es lo más habitual, vamos a usarla como base para implementar nuestros procesadores:

AbstractProcessor – Implementación por defecto de los métodos de Processor	
Método	Descripción
<code>getSupportedAnnotationTypes</code>	Devuelve el valor de la anotación <code>@SupportedAnnotationTypes</code> . Por defecto: conjunto vacío de tipos anotación soportados.
<code>getSupportedOptions</code>	Devuelve el valor de la anotación <code>@SupportedOptions</code> . Por defecto: conjunto vacío de opciones soportadas.
<code>getSupportedSourceVersion</code>	Devuelve el valor de la anotación <code>@SupportedSourceVersion</code> . Por defecto: <code>SourceVersion</code> de la versión de la JDK en ejecución.
<code>init</code>	Inicializa el procesador de anotaciones. IMPORTANTE : guarda una instancia del entorno de procesamiento <code>ProcessingEnvironment</code> en la variable de instancia protegida <code>processingEnv</code> de la clase. Esta variable de instancia será fundamental para la implementación del procesamiento. No se puede llamar a <code>init</code> más de una vez o se arrojará una excepción.
<code>process</code>	Método de procesamiento. Se deja <code>abstract</code> para que sean las subclases las que implementen una lógica de procesamiento concreta.
<code>getCompletions</code>	Devuelve una lista de sugerencias vacía.

Ejemplo: Proc. anotaciones básico heredando de AbstractProcessor

El método estándar para implementar un procesador JSR 269 es heredando de **AbstractProcessor**, anotando la clase con las anotaciones `@SupportedAnnotationTypes`, `@SupportedOptions` y `@SupportedSourceVersion`, y redefiniendo los métodos oportunos.

El procesador JSR 269 más básico que tiene sentido implementar, que únicamente arroja mensajes informativos a la salida del compilador, tendría el siguiente aspecto:

```
@SupportedAnnotationTypes("*)
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class HolaMundoProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> tiposAnot, RoundEnvironment roundEnv) {

        // PROCESAMIENTO SEGÚN LA RONDA EN LA QUE NOS ENCONTREMOS

        if (!roundEnv.processingOver()) {

            // RONDA NO FINAL

            // imprimimos un mensaje informativo por la salida del compilador
            this.processingEnv.getMessager().printMessage(Kind.NOTE, "¡Hola, mundo!");

        } else {

            // RONDA FINAL

            // simplemente presentamos un mensaje informativo para
            // mostrar que el procesador ha finalizado su procesamiento
            this.processingEnv.getMessager().printMessage(Kind.NOTE, "¡Adiós, mundo!");

        }

        // finalizamos retornando false porque no queremos quedarnos con el procesado
        // de este tipo anotación en exclusiva; si devolvemos true el conjunto de
        // anotaciones que hemos procesado no será pasado a otros procesadores
        return false;
    }
}
```

Cuando es ejecutado, en la primera ronda escribe ¡Hola, mundo! y en la ronda final de procesamiento vuelve a ser llamado y escribe ¡Adiós, mundo!:

```
Note: ¡Hola, mundo!
Note: ¡Adiós, mundo!
```

Nótese como en el cuerpo de `process` se hace referencia a la variable del entorno de procesamiento `ProcessingEnvironment` llamada `processingEnv`. Recordemos que es una variable de instancia de visibilidad protegida de la cual podremos obtener todas las facilidades fundamentales para realizar el procesamiento. En el código se ve, por ejemplo, cómo se accede a una instancia de `Messager` a través de `getMessager()` para emitir los mensajes informativos.

Este es un ejemplo para ilustrar lo mínimo que se le exige tener a un procesador de anotaciones, pero, obviamente, no es un ejemplo práctico. Normalmente, los procesadores, incluso para realizar tareas sencillas, guardarán como variables de instancia referencias, además de al entorno de procesamiento, a otras instancias, como un `Messager` para emitir mensajes de compilación. El siguiente ejemplo es una ilustración más completa y ajustada a la realidad práctica habitual de lo que suele contener el cuerpo de un procesador de anotaciones JSR 269.

Ejemplo: Proc. anotaciones típico heredando de AbstractProcessor

A continuación presentamos el código de un procesador de anotaciones JSR 269 más prototípico de un caso real. Aquí se ve cómo se mantienen algunas variables de instancia para acceder a las facilidades de procesamiento de forma más cómoda, o cómo se redefine el método `init` para inicializar el procesador con dichas variables de instancia a partir de las instancias devueltas por el entorno de procesamiento `processingEnv`.

La variable de instancia `processingEnv` estará disponible justo a partir de la llamada a la implementación del método de inicialización de la superclase `super.init(env)`, que hay que hacer siempre y que, en este caso, se encargará de asignar el valor de `processingEnv` para que podamos utilizarla. De ella, podremos extraer los objetos que implementan las facilidades más básicas para poder llevar a cabo el procesamiento de anotaciones, como el objeto de mensajería (`Messager`), el de escritura a ficheros (`Filer`), o los de utilidades de elementos (`Elements`) y utilidades de tipos (`Types`), entre otros:

```
@SupportedAnnotationTypes("anotaciones.procesamiento.java67.tipico.AnotacionEjemploTipico")
@SupportedOptions({})
@SupportedSourceVersion(SourceVersion.RELEASE_7)
public class AnotacionEjemploTipicoProcessor extends AbstractProcessor {

    // =====
    // VARIABLES DE INSTANCIA
    // =====

    // objeto de mensajería informativa (lo definimos en una vrble por comodidad)
    private Messager messenger;

    // objeto de escritura a ficheros (lo definimos en una vrble por comodidad)
    private Filer filer;

    // objeto de utilidades de elementos (lo definimos en una vrble por comodidad)
    private Elements elements;

    // objeto de utilidades de tipos (lo definimos en una vrble por comodidad)
    private Types types;

    // =====
    // MÉTODOS DE APLICACIÓN
    // =====

    // -----
    // INICIALIZACIÓN
    // -----

    @Override
    public void init(ProcessingEnvironment env) {

        // llamamos al método de la super-clase por
        // si ejecutara algún código importante que
        // no deje de ejecutarlo al redefinir este método
        super.init(env);

        // en este método de inicialización aprovechamos
        // para asignar las variables de instancia privadas
        // que nos puedan venir bien por comodidad
        this.messenger = this.processingEnv.getMessager();
        this.filer = this.processingEnv.getFiler();
        this.elements = this.processingEnv.getElementUtils();
        this.types = this.processingEnv.getTypeUtils();
    }
}
```

```

// -----
// PROCESAMIENTO
// -----

@Override
public boolean process(Set<? extends TypeElement> tiposAnot, RoundEnvironment roundEnv) {

    // PROCESAMIENTO SEGÚN LA RONDA EN LA QUE NOS ENCONTREMOS

    if (roundEnv.processingOver()) {

        // RONDA FINAL

        // imprimimos un mensaje informativo por la salida del compilador
        this.processingEnv.getMessager().printMessage(Kind.NOTE,
            "Ronda final. Procesamiento finalizado.");

    } else {

        // RONDA NO FINAL

        // ITERACIÓN SOBRE LOS ELEMENTOS RAÍZ A PROCESAR

        // iteramos por los elementos raíz a procesar
        for (Element elem : roundEnv.getRootElements()) {

            // PROCESAMIENTO DEL ELEMENTO

            // ... CUERPO DEL PROCESAMIENTO DE ANOTACIONES ...
        }
    }

} // process
}

```

También queda reflejado lo que suele ser la estructura de procesamiento de elementos más básica que puede implementarse en **process**: un bucle **for** que recorra todos los elementos raíz que hay que procesar en la actual ronda de procesamiento.

IMPORTANTE: Recordemos que si **roundEnv.processingOver()** es **true** es porque nos encontramos en la “ronda final” del procesamiento. En dicho caso, lo más habitual es que el conjunto de tipos anotación a procesar (**tiposAnot**) llegue vacio. Pero lo más importante que hay que tener en cuenta es que tendremos que construir la lógica del método **process** teniendo en cuenta que dicha lógica debería ser correcta para el caso de la ronda final. En el ejemplo se realiza una acción específicamente para la ronda final, que es imprimir un mensaje informativo, pero también es muy común ver procesadores que no hacen nada en la ronda final, poniendo una guarda para el procesamiento con un **if** que se asegura de no estar en la ronda final:

```

if (!roundEnv.processingOver()) { // si NO estamos en la ronda final (nótese el !)

    // ... PROCESAMIENTO DE ANOTACIONES ...
}

```

La instancia del entorno de la ronda de procesamiento **RoundEnvironment**, además de ofrecernos información sobre el estado de la ronda de procesamiento actual, será también el que nos proporcione la información necesaria para localizar o “descubrir” los elementos de código fuente Java que tenemos que procesar. En el siguiente subapartado vamos a explicar en detalle los distintos métodos posibles para el “descubrimiento” de las anotaciones que deberemos procesar en nuestro procesador de anotaciones.

15.3.1.2.- Descubrimiento de elementos a procesar.

Lo más habitual al implementar un procesador de anotaciones en el método **process** es empezar teniendo que localizar o, más concretamente, se suele utilizar el término “descubrir” los elementos de código que dicho procesador debe procesar.

Cuando se invoca el procesamiento de anotaciones, se le da un conjunto de clases sobre las que tiene que operar. Además, al principio de cada ronda se establecen los denominados “**elementos raíz**” (“root elements”). Se llaman los elementos “raíz” porque son los elementos de donde parte el procesamiento, y que darán lugar, como si fueran la raíz de un árbol que está a punto de crecer, a otros elementos para ser procesados y que serán como el cuerpo del árbol que ha crecido a partir de su raíz. Los elementos raíz de la primera ronda serán los tipos dados por los ficheros **.java** (**package-info.java** incluidos) sobre los que se invoca el procesamiento.

Los “elementos raíz” darán lugar a los denominados “**elementos incluidos**” (“*included elements*”), que incluirán a los propios “elementos raíz”, además todos los elementos anidados que tengan dentro de ellos: miembros, parámetros o parámetros de tipo.

Para poder descubrir los elementos a procesar dispondremos de los siguientes **métodos de descubrimiento de elementos** pertenecientes a la interfaz **RoundEnvironment**:

1) **Set<? extends Element> getRootElement()**

getRootElement devuelve el conjunto de “elementos raíz” de la ronda actual. Es lo mínimo que necesitaremos para empezar a buscar los elementos anotados con el tipo anotación a procesar. A partir del conjunto de “elementos raíz”, el procesador podrá construir la lógica para ir recorriendo cada uno de ellos, examinando si es oportuno sus distintos miembros (campos, métodos, constructores, clases anidadas, etcétera) y pudiendo así descubrir cuáles están anotados con el tipo anotación a procesar.

```
@Override
public boolean process(Set<? extends TypeElement> tiposAnotacion, RoundEnvironment roundEnv) {

    // ITERACIÓN SOBRE LOS ELEMENTOS RAÍZ A PROCESAR

    // iteramos por los elementos raíz a procesar
    for (Element elem : roundEnv.getRootElement()) {

        // COMPROBACIÓN: ELEMENTO ANOTADO CON EL TIPO ANOTACIÓN A PROCESAR

        if (elem.getAnnotation(AnotacionEjemploTipico.class) == null) {

            // si el elemento no está anotado
            // con el tipo anotación a procesar -> lo ignoramos
            continue;
        }

        // PROCESAMIENTO DEL ELEMENTO ANOTADO

        // ... código del procesamiento ...
    }

    // finalizamos retornando false porque no queremos quedarnos con el procesado
    // de este tipo anotación en exclusiva; si devolvemos true el conjunto de
    // anotaciones que hemos procesado no será pasado a otros procesadores
    return false;
} // process
```

```
2) Set<? extends Element> getElementsAnnotatedWith(TypeElement a)
0 Set<? extends Element> getElementsAnnotatedWith(Class<? extends Annotation> a)
```

getElementsAnnotatedWith toma como argumento un tipo anotación y nos devuelve los elementos anotados con ese tipo anotación en la ronda actual. El tipo anotación argumento del método se puede pasar como elemento de tipo anotación, o como el **Class** de la anotación como instancia de la interfaz **java.lang.annotation.Annotation** de la API de reflexión.

getElementsAnnotatedWith tiene la ventaja de que devuelve los elementos anotados con un tipo anotación entre los “elementos incluidos” para su procesamiento en la presente ronda. Se devuelven todos los elementos anotadas con el tipo anotación parámetro de cualquier tipo en un orden indeterminado. Este método es muy interesante, ya que, nos ahorra el tener que “navegar” por la jerarquía de elementos buscando los elementos anotados a procesar.

getElementsAnnotatedWith vendrá bien para el procesamiento de tipos anotación cuyas anotaciones puedan ser procesadas en un orden indeterminado, lo cual no siempre es posible. Por ejemplo, puede ocurrir que la lógica de procesamiento de los subelementos de una clase esté vinculada a su clase contenedora y, debido a ello, necesitaremos forzosamente ir procesando las anotaciones en bloques de clase por clase siguiendo un orden determinado.

```
@Override
public boolean process(Set<? extends TypeElement> tiposAnotacion, RoundEnvironment
roundEnv) {

    // ITERACIÓN SOBRE LOS ELEMENTOS ANOTADOS CON EL TIPO ANOTACIÓN A PROCESAR

    // iteramos sobre los elementos anotados con el tipo anotación a procesar
    for (Element elem : roundEnv.getElementsAnnotatedWith(AnotacionEjemploTipico.class)) {

        // COMPROBACIÓN: ELEMENTO ANOTADO CON EL TIPO ANOTACIÓN A PROCESAR

        // NO es necesaria, gracias a getElementsAnnotatedWith tenemos garantizado
        // que los elem. devueltos están anotados con el tipo anotación a procesar

        // PROCESAMIENTO DEL ELEMENTO ANOTADO

        // ... código del procesamiento ...
    }

    // finalizamos retornando false porque no queremos quedarnos con el procesado
    // de este tipo anotación en exclusiva; si devolvemos true el conjunto de
    // anotaciones que hemos procesado no será pasado a otros procesadores
    return false;
}
```

Otra forma de utilizar `getElementsAnnotatedWith` para descubrir elementos de código anotados tiene que ver con procesadores que, en lugar de procesar un tipo anotación único, puede procesar varios tipos anotación diferentes.

Un procesador de anotaciones soporte dos o más tipos anotación, en lugar usar el `.class` del tipo anotación (ejemplo anterior: `getElementsAnnotatedWith(AnotacionEjemploTipico.class)`), puede iterar sobre el parámetro `tiposAnotacion` de los tipos anotación a procesar que nos pasa la herramienta de procesamiento, y utilizar la información de `getElementsAnnotatedWith` para obtener en cada iteración los elementos anotados con cada uno de dichos tipos anotación. Véase, por ejemplo, el siguiente esquema de procesamiento:

```
@Override
public boolean process(Set<? extends TypeElement> tiposAnotacion, RoundEnvironment roundEnv) {

    // ITERACIÓN SOBRE LOS TIPOS ANOTACIÓN A PROCESAR

    // en lugar de escribir directamente el .class del tipo anotación a procesar,
    // ya sea en el caso de un procesador de anotaciones que procese tanto uno
    // como especialmente varios tipos anotación, podemos escoger montar un for
    // adicional para iterar sobre los diferentes tipos anotación a procesar
    // que nos llegan en el parámetro tiposAnotacion del método process

    for (TypeElement tipoAnotacion : tiposAnotacion) {

        // ITERACIÓN SOBRE LOS ELEMENTOS ANOTADOS CON EL TIPO ANOTACIÓN A PROCESAR

        // iteramos sobre los elementos anotados con el tipo anotación a procesar
        for (Element elem : roundEnv.getElementsAnnotatedWith(tipoAnotacion)) {

            // COMPROBACIÓN: ELEMENTO ANOTADO CON EL TIPO ANOTACIÓN A PROCESAR

            // NO es necesaria, gracias a getElementsAnnotatedWith tenemos garantizado
            // que los elementos devueltos están anotados con el tipo anotación a procesar

            // PROCESAMIENTO DEL ELEMENTO ANOTADO

            // ... código del procesamiento ...
        }
    }

    // finalizamos retornando false porque no queremos quedarnos con el procesado
    // de este tipo anotación en exclusiva; si devolvemos true el conjunto de
    // anotaciones que hemos procesado no será pasado a otros procesadores
    return false;
}
```

El caso de procesadores de anotaciones que procesen varios tipos anotación no es común, pero tampoco es raro verlos, como en el caso de APIs de validación como [Hibernate Validator](#), cuyo [procesador](#) (`org.hibernate.validator.ap.ConstraintValidationProcessor`), valida todos los diferentes tipos anotación soportados por dicha API él solo.

En base a los requisitos de la lógica a implementar y, sobre todo, de los elementos de código sobre los cuales el tipo anotación va a ser aplicado, los desarrolladores de procesadores tendrán que elegir en cada caso una “**estrategia de descubrimiento de anotaciones**”. Debiendo elegirse aquella estrategia que, teniendo siempre una **cobertura completa de las anotaciones que se desean procesar**, resulte más eficiente y con una estructura bien separada y cómoda de mantener, para lo cual los Visitor pueden ser útiles en ciertas ocasiones.

IMPORTANTE: Los métodos anteriores no devuelven las anotaciones que se encuentran dentro del cuerpo de los métodos. Para descubrir esas anotaciones podemos usar las facilidades de la Compiler Tree API. Véase el subapartado siguiente.

15.3.1.3.- Descubrimiento de elementos usando Compiler Tree API.

Gracias a la Compiler Tree API podemos descubrir anotaciones que no nos ofrecen los métodos de descubrimiento de la interfaz `RoundEnvironment` de la API JSR 269.

Instanciación de un objeto `Trees`

Lo primero que deberemos hacer es instanciar un objeto de la clase `Trees` del paquete `com.sun.source.util`. Este objeto nos proporcionará métodos de conveniencia muy útiles para traspasar información entre la Compiler Tree API y las APIs JSR 269. Para instanciar `Trees` necesitamos una instancia preexistente de una `JavaCompiler.CompilationTask` de la Java Compiler API o de `ProcessingEnvironment` de la API de procesamiento JSR 269. En nuestro caso, lo más fácil es esta última opción, ya que el procesador de anotaciones tiene acceso directo al entorno de procesamiento. La instanciación de la variable `Trees` la realizamos en el método `init` al inicializar el constructor para tenerla disponible desde el principio:

```
@Override  
public void init(ProcessingEnvironment env) {  
  
    // ... resto del código ...  
  
    // objeto de árboles sintácticos  
    this.trees = Trees.instance(this.processingEnv);  
}
```

`Trees.getElement`: su importancia y sus limitaciones

Utilizar la Compiler Tree API para descubrir anotaciones es posible, además de gracias a la posibilidad de instanciar `Trees` a partir de una instancia de `ProcessingEnvironment`, básicamente a la existencia del siguiente método de la clase abstracta `Trees`:

```
public abstract Element getElement(TreePath path)
```

El método `getElement` es trascendental, ya que nos permitirá obtener un `Element` de la API JSR 269 a partir de una instancia de la clase `TreePath` de la Compiler Tree API, permitiendo de esta manera un cierto grado de interoperabilidad entre ambas APIs.

IMPORTANTE: Desgraciadamente para nosotros, **no todos los tipos de nodos dados por `TreePath` están soportados para obtener de ellos un `Element`**. Uno de los casos que hemos podido probar durante la confección de los ejemplos de código para el presente manual es el de los bloques inicializadores, ya sean estáticos o dinámicos, modelados en la Model API como `ElementKind.STATIC_INIT` y `ElementKind.INSTANCE_INIT` respectivamente, y que pueden ser descubiertos por la Compiler Tree API como instancias de la interfaz `BlockTree` de la misma. No obstante, cuando se pasa su `TreePath` correspondiente al método `getElement`, este nos devuelve `null`. Una pena, ya que, debido a esto, no es posible trabajar con elementos de bloques inicializadores en un procesador de anotaciones JSR 269. Aún así, no es que esto sea un impedimento en absoluto importante, ya que **el compilador Java no soporta la colocación de anotaciones sobre los bloques inicializadores**. La Model API los modela como elementos, pero recordemos que la finalidad de la Model API es modelar el lenguaje, no sólo servir como API de base para el procesamiento de anotaciones.

Descubrimiento de variables dentro de los métodos y constructores

Ninguno de los métodos de descubrimiento de `RoundEnvironment` que hemos visto en el subapartado anterior, ni `getRootElement`s ni `getElementsAnnotatedWith`, devuelven los elementos anotados situados dentro de los cuerpos de los métodos. No obstante, con la Compiler Tree API sí podremos extraer esa información si, por ejemplo, montamos un Visitor que sea capaz de recorrer la estructura sintáctica interna de los cuerpos de los métodos y crear a partir de ahí los `Element`s de las variables encontradas.

Para ello, en nuestro caso, hemos utilizado el Visitor `TreePathScanner` de la Compiler Tree API para recorrer las estructuras y subestructuras sintácticas a partir de un nodo raíz dado:

```
private static class VariablesTreePathScanner extends TreePathScanner<Void, Void> {

    // =====
    // VARIABLES DE INSTANCIA
    // =====

    public CompilationUnitTree unidadCompilacion;
    public Trees trees;
    public ArrayList<VariableElement> variables = new ArrayList<VariableElement>();

    // =====
    // CONSTRUCTOR
    // =====

    public VariablesTreePathScanner(CompilationUnitTree unidadCompilacion, Trees trees) {
        this.unidadCompilacion = unidadCompilacion;
        this.trees = trees;
    }

    // =====
    // MÉTODOS DE APLICACIÓN
    // =====

    @Override
    public Void visitVariable(VariableTree variableTree, Void param) {
        // TreePath correspondiente a la variable
        TreePath variableTreePath = this.trees.getPath(unidadCompilacion, variableTree);

        // Element de la variable
        VariableElement variableElem =
            (VariableElement) this.trees.getElement(variableTreePath);

        // añadimos la variable a la lista
        variables.add(variableElem);

        // llamamos al método de la superclase para que se visiten
        // los elementos sintácticos descendientes de este
        return super.visitVariable(variableTree, param);
    }
}
```

Este Visitor rastreará todas las construcciones sintácticas de una rama del árbol sintáctico empezando por el nodo raíz que nos convenga y, dentro de dicho árbol sintáctico, convertirá todas las construcciones sintácticas de variables (interfaz `VariableTree`) en elementos JSR 269 `VariableElement` gracias al método `getElement`. Las variables descubiertas se guardan en la lista `variables` del Visitor para ser recuperadas cuando este haya terminado su rastreo y así poder procesar dichas variables descubiertas en el procesamiento de anotaciones estándar.

Usando el Visitor **VariablesTreePathScanner** podemos, por tanto, definir un método auxiliar que nos permitirá, a partir de un elemento **Element** de JSR 269 y utilizando dicho Visitor, recuperar la lista de elementos de tipo **VariableElement** que dicho elemento contiene en su interior. Sin embargo, una vez descubiertas dichas variables, excluiremos de entre ellas las de tipo parámetro (**ElementKind.PARAMETER**), puesto que esas sí son directamente accesibles a través del método **getParameters** dado por la interfaz **ExecutableElement**. El código de dicho método auxiliar **descubrirVariablesNoParametro** es el siguiente:

```

private ArrayList<VariableElement> descubrirVariablesNoParametro(Element elemento) {

    // DESCUBRIMIENTO DE VARIABLES NO PARÁMETRO (VÍA Compiler Tree API)

    // las variables de tipo parámetro se pueden recuperar a través
    // de ExecutableElement.getParameters(); no así las demás;
    // las variables no parámetro (variables locales, los parámetros
    // de los manejadores de excepciones y las vrbles de recurso)
    // tenemos que buscarlas por nuestra cuenta, ya que la API JSR 269
    // no nos las proporciona por ningún método de descubrimiento;
    // por eso aquí vamos a usar para ello las facilidades de la
    // Compiler Tree API, que nos permite recorrer el árbol sintáctico
    // de un elemento y extraer de él todos los elementos de variables

    // vrble resultado
    ArrayList<VariableElement> variablesNoParametro = new ArrayList<VariableElement>();

    // comprobación de nulidad
    if (elemento == null) return variablesNoParametro;

    // DESCUBRIMIENTO DE VARIABLES

    // recuperamos las variables del elemento
    ArrayList<VariableElement> variables = new ArrayList<VariableElement>();

    // referencia al elemento padre
    Element elementoPadre = elemento.getEnclosingElement();

    // comprobación de nulidad
    if (elementoPadre == null) return variablesNoParametro;

    // si el elemento padre es una clase o un tipo enumerado
    // (si es interfaz o tipo anotación no puede tener variables
    // no parámetro en sus métodos) examinamos su árbol sintáctico
    // en busca de variables a través de un TreePathScanner
    if ((elementoPadre.getKind() == ElementKind.CLASS)
        || (elementoPadre.getKind() == ElementKind.ENUM)) {

        // TreePath del elemento a partir de la cual se va a escanear
        TreePath elementoTreePath = this.trees.getPath(elemento);

        // comprobación de nulidad
        if (elementoTreePath == null) return variablesNoParametro;

        // unidad de compilación (la necesitamos para el scanner)
        CompilationUnitTree unidadCompilacion = elementoTreePath.getCompilationUnit();

        // instanciamos el TreePathScanner
        VariablesTreePathScanner variablesScanner =
            new VariablesTreePathScanner(unidadCompilacion, this.trees);

        // lanzamos el scanner de variables
        variablesScanner.scan(elementoTreePath, null);

        // una vez finalizado el escaneo,
        // recuperamos la lista de variables
        variables = variablesScanner.variables;
    }
}

```

```

// FILTRADO DE VARIABLES PARÁMETRO

for (VariableElement variable : variables) {

    // IMPORTANTE: sólo si la variable es no es de tipo parámetro;
    // ignoramos los parámetros de métodos, ya que estas variables
    // sí pueden recuperarse a través de ExecutableElement.getParameters()

    // añadimos el elemento de la variable a la lista (si procede)
    if (variable.getKind() != ElementKind.PARAMETER) {

        // añadimos la variable no parámetro a la lista
        variablesNoParametro.add(variable);
    }
}

// finalmente, devolvemos el resultado
return variablesNoParametro;
}

```

IMPORTANTE: El método anterior nos permite sortear una de las mayores limitaciones de la API de procesamiento de anotaciones JSR 269: la incapacidad de poder procesar anotaciones sobre variables locales y otros elementos dentro del cuerpo de un método.

NOTA: Para ver el ejemplo de uso completo de esta funcionalidad, véase el código fuente del primer ejemplo de procesador JSR 269 “**Navegación básica: @ImprimirMensaje**”. En dicho ejemplo se ha tratado de descubrir anotaciones sobre el mayor número posible de tipos de elementos. Donde no era posible descubrir las anotaciones mediante la API JSR 269, se ha utilizado la Compiler Tree API. Esto ha permitido descubrir anotaciones sobre variables locales, parámetros de manejadores de excepción y variables de recurso (introducidas en Java SE 7), tanto en los cuerpos de los métodos y constructores, como en el interior de los bloques inicializadores, tanto estáticos como de instancia. Lo siguiente es sólo un extracto de su salida:

```

D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java67\src\main\java\anotaciones\procesamiento\java67\ejemplos\imprimirMensaje\elementosAnotados\Im-
primirMensajeClaseAnotada.java:153: Note: VARIABLE LOCAL ANOTADA
    String variableLocal = "Mensaje: ";
               ^
Note: VARIABLE LOCAL: variableLocal
Note: MENSAJE: Variable local.
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java67\src\main\java\anotaciones\procesamiento\java67\ejemplos\imprimirMensaje\elementosAnotados\Im-
primirMensajeClaseAnotada.java:155: Note: VARIABLE DE RECURSO ANOTADA
    try (@ImprimirMensaje("Variable de recurso.")) BufferedWriter escritorFichero =
               ^
Note: VARIABLE DE RECURSO: escritorFichero
Note: MENSAJE: Variable de recurso.
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java67\src\main\java\anotaciones\procesamiento\java67\ejemplos\imprimirMensaje\elementosAnotados\Im-
primirMensajeClaseAnotada.java:161: Note: PARAM EXCEP ANOTADO
    } catch (@ImprimirMensaje("Parámetro de manejador de excepción.")) IOException e {
               ^
Note: PARAM EXCEP: e
Note: MENSAJE: Parámetro de manejador de excepción.

```

15.3.1.4.- Resumen de resultados del proceso de descubrimiento.

Como hemos visto en el apartado anterior, la API JSR 269 tiene una importante limitación a la hora de descubrir anotaciones: no permite descubrir ninguna anotación ubicada dentro de los cuerpos de los métodos.

Además de lo anterior, hay otra serie de consideraciones a tener en cuenta a la hora de tratar con los diferentes elementos JSR 269 o con los **Tree** de la Compiler Tree API, como si dichas APIs modelan ciertos elementos o si son capaces de descubrirlos.

Finalmente se ha de tener en cuenta si los diferentes Tree de la Compiler Tree API son convertibles en elementos **Element** de JSR 269 a través del método **Trees.getElement**.

La siguiente tabla resume toda esta información en base a lo que han sido los resultados de nuestras pruebas de uso de ambas APIs:

Resultados del proceso de descubrimiento de anotaciones según su ubicación						
Ubicación de la anotación	javac	JSR 269		Compiler Tree API		
	¿Compila? (Java SE 6/7)	¿Elem. modelado? (ElementKind)	¿Elemento descubrible?	¿Árbol modelado? (Tree.Kind)	¿Árbol descubrible?	¿Convertible en elemento?
Paquete	SÍ	PACKAGE	SÍ	COMPILATION_UNIT	SÍ (manual)	SÍ
Clase	SÍ	CLASS	SÍ	CLASS	SÍ (manual)	SÍ
Enumerado	SÍ	ENUM	SÍ	ENUM	SÍ (manual)	SÍ
Interfaz	SÍ	INTERFACE	SÍ	INTERFACE	SÍ (manual)	SÍ
Tipo anotación	SÍ	ANNOTATION_TYPE	SÍ	ANNOTATION_TYPE	SÍ (manual)	SÍ
Constructor	SÍ	CONSTRUCTOR	SÍ	METHOD	SÍ (manual)	SÍ
Método	SÍ	METHOD	SÍ	METHOD	SÍ (manual)	SÍ
Campo	SÍ	FIELD	SÍ	VARIABLE	SÍ (manual)	SÍ
Constante de un tipo enumerado	SÍ	ENUM_CONSTANT	SÍ	VARIABLE	SÍ (manual)	SÍ
Parámetro	SÍ	PARAMETER	SÍ	VARIABLE	SÍ (manual)	SÍ
Variable local	SÍ	LOCAL_VARIABLE	NO	VARIABLE	SÍ (manual)	SÍ
Parám. de excepción	SÍ	EXCEPTION_PARAMETER	NO	VARIABLE	SÍ (manual)	SÍ
Variable de recurso (Java SE 7+)	SÍ	RESOURCE_VARIABLE	NO	VARIABLE	SÍ (manual)	SÍ
Parámetro de tipo	NO	TYPE_PARAMETER	NO	TYPE_PARAMETER	SÍ (manual)	NO
Inicializador estático	NO	STATIC_INIT	NO	BLOCK	SÍ (manual)	NO
Inicializador de instancia	NO	INSTANCE_INIT	NO	BLOCK	SÍ (manual)	NO

Estos resultados vienen a refrendar lo que se ha comentado en los apartados anteriores: que la API JSR 269 no permite descubrir los elementos de código ubicados dentro de los cuerpos de los métodos (como variables locales o de recurso y los parámetros de excepción).

Asimismo, los parámetros de tipo y los bloques inicializadores (ya sean estáticos o de instancia) se pueden descubrir a través de la Compiler Tree API, pero no se pueden pasar a **Element** usando **Trees.getElement** y, de hecho, el compilador no compilará ningún código que trate de poner anotaciones sobre ellos.

15.3.1.5.- Opciones para los procesadores desde línea de comandos.

Los procesadores de anotaciones pueden implementar un sinfín de acciones y, a su vez, dichas acciones podrían ofrecer diversas opciones de configuración. Por este motivo, existe un mecanismo para pasar opciones de configuración a los procesadores de anotaciones.

Los procesadores de anotaciones, como vimos en el subapartado dedicado a ellos, pueden declarar las opciones que soportan a través del método `getSupportedOptions()` de la interfaz `Processor` o de la anotación `@SupportedOptions` anotando un `AbstractProcessor`.

Luego, la forma de establecer valor para dichas opciones soportadas por los procesadores será pasando dichos valores a través de las opciones `-A` de `javac`, según el siguiente formato:

`-Aclave [=valor]`

Las opciones que comienzan por `-A` son para los procesadores de anotaciones. No son interpretadas por `javac`, si no que se ponen a disposición de los procesadores a través del método `Map<String, String> getOptions()` de `ProcessingEnvironment`.

Supongamos, por ejemplo, que declaramos que nuestro procesador soporta las opciones `@SupportedOptions({ "opcion.valor.string", "opcion.valor.integer" })`. Si invocamos a `javac` estableciendo los valores de dichas opciones según el formato esperado:

`-Aopcion.valor.string="¡Hola, mundo!" -Aopcion.valor.integer=12345`

NOTA: El valor de la propiedad `string`, al contener espacios, se ha delimitado entre comillas “”.

El siguiente código nos permitirá recuperar los valores de dichas opciones:

```
// referencia a las opciones del procesador
Map<String, String> opciones = this.processingEnv.getOptions();

// valor de la opción: opcion.valor.string
String opcionValorString = opciones.get("opcion.valor.string");
this.message.printMessage(Kind.NOTE,
    "procesador: opciones: opcion.valor.string = " + opcionValorString);

// valor de la opción: opcion.valor.integer
String opcionValorInteger = opciones.get("opcion.valor.integer");
this.message.printMessage(Kind.NOTE,
    "procesador: opciones: opcion.valor.integer = " + opcionValorInteger);
```

La salida que produce la ejecución de este código es la salida correcta esperada:

```
Note: procesador: opciones: opcion.valor.string = ¡Hola, mundo!
Note: procesador: opciones: opcion.valor.integer = 12345
```

La `opcion.valor.integer` representa un número y requiere de un valor numérico. El parseo y validación de los valores dados para las opciones será también parte de la lógica de los procesadores de anotaciones. Por ejemplo, si se pasara un valor ilegal que no representara ningún valor numérico, el procesador de anotaciones debería arrojar un error de compilación con un mensaje de error significativo para poder identificar y corregir el valor incorrecto.

Ejemplo: Paso de opciones desde línea de comandos con valores por defecto y validación

Vamos a realizar un ejemplo de paso de opciones desde línea de comandos con la lógica más habitual que se estilará en esos casos: comprobación de si las opciones han sido o no establecidas y, en caso negativo, darles un valor por defecto, así como la correcta validación de los valores establecidos en el caso de que una de las opciones tenga valor numérico.

El siguiente código recupera los valores de las opciones soportadas por el procesador, los analiza para ver si es necesario establecer los valores por defecto o si hay que parsear un valor numérico, y finalmente muestra los valores finales de dichas opciones.

```
// OBTENCIÓN DE LOS VALORES DE LAS OPCIONES SOPORTADAS

// referencia a las opciones del procesador
Map<String, String> opciones = this.processingEnv.getOptions();

// valor de la opción: opcion.valor.string
String opcionValorString = opciones.get("opcion.valor.string");
this.messager.printMessage(Kind.NOTE,
    "procesador: opciones: opcion.valor.string (leida) = " + opcionValorString);

// valor de la opción: opcion.valor.integer
String opcionValorInteger = opciones.get("opcion.valor.integer");
this.messager.printMessage(Kind.NOTE,
    "procesador: opciones: opcion.valor.integer (leida) = " + opcionValorInteger);

// VALIDACIÓN DE LOS VALORES DE LAS OPCIONES SOPORTADAS

// opcionValorString: comprobamos si es null o cadena vacía,
// en cuyo caso se establecerá el valor por defecto
if (opcionValorString==null || opcionValorString.isEmpty()) {
    opcionValorString = "¡Hola, mundo!"; // valor por defecto
}

// opcionValorInteger: comprobamos si es null o cadena vacía,
// en cuyo caso se establecerá el valor por defecto;
// si no, parseamos el valor dado a número y arrojamos
// una excepción si no fuera un valor entero válido
int opcionValorIntegerFinal = 1; // valor por defecto
if (opcionValorInteger!=null && !opcionValorInteger.isEmpty()) {
    try {
        opcionValorIntegerFinal = Integer.parseInt(opcionValorInteger);
    } catch (NumberFormatException nfe) {
        this.messager.printMessage(Kind.ERROR,
            "procesador: opciones: opcion.valor.integer: valor incorrecto: " + nfe);
    }
}

// INFORMACIÓN DE LOS VALORES FINALES DE LAS OPCIONES SOPORTADAS

// mostramos como mensaje informativo (que puede ser muy útil para depuración)
// los valores finales de las opciones soportadas por el procesador
this.messager.printMessage(Kind.NOTE,
    "procesador: opciones: opcion.valor.string (final) = " + opcionValorString);
this.messager.printMessage(Kind.NOTE,
    "procesador: opciones: opcion.valor.integer (final) = " + opcionValorIntegerFinal);

// ... resto del procesamiento ...
```

Ahora, si se ejecuta el procesamiento sin establecer valor para las opciones, el procesador nos muestra la salida correcta esperada, asignando sus valores por defecto como valor final:

```
Note: procesador: opciones: opcion.valor.string (leida) = null
Note: procesador: opciones: opcion.valor.integer (leida) = null
Note: procesador: opciones: opcion.valor.string (final) = ¡Hola, mundo!
Note: procesador: opciones: opcion.valor.integer (final) = 1
```

Si se ejecuta el procesamiento estableciendo valores para las opciones, el procesador nos muestra la salida correcta esperada, asignando a cada opción los valores establecidos:

```
Note: procesador: opciones: opcion.valor.string (leida) = ¡Hola, universo!
Note: procesador: opciones: opcion.valor.integer (leida) = 123456789
Note: procesador: opciones: opcion.valor.string (final) = ¡Hola, universo!
Note: procesador: opciones: opcion.valor.integer (final) = 123456789
```

Finalmente, si como valor de `opcion.valor.integer` no se establece un valor numérico correcto, por ejemplo: `-Aopcion.valor.integer="Valor alfanumérico"`, obtenemos también el comportamiento esperado: se arroja la excepción `NumberFormatException` y se envía al compilador un mensaje de error significativo:

```
Note: procesador: opciones: opcion.valor.string (leida) = ¡Hola, universo!
Note: procesador: opciones: opcion.valor.integer (leida) = Valor alfanumérico
error: procesador: opciones: opcion.valor.integer: valor incorrecto:
java.lang.NumberFormatException: For input string: "Valor alfanumérico"
Note: procesador: opciones: opcion.valor.string (final) = ¡Hola, universo!
Note: procesador: opciones: opcion.valor.integer (final) = 1
```

El provocar un error de compilación detiene todo el proceso, y así el error del valor de la opción numérica podrá identificarse y corregirse. Nótese además como, con la lógica construida en el código de ejemplo, puesto que el valor dado para `opcion.valor.integer` era incorrecto, se ha establecido a su valor por defecto, incluso tras fallar el parseo. Esto es síntoma de que se ha realizado un tratamiento de errores bien controlado y se ha establecido un mecanismo correcto de valores por defecto.

El que incluso en una condición de error la opción se establezca correctamente a un valor por defecto no quita que, al no estar pasándose un valor correcto, la lógica del procesador opte por emitir un error de compilación. Ignorar un valor erróneo de configuración y asumir un valor por defecto siempre es peligroso, ya que el usuario que ha invocado el procesamiento de anotaciones puede no estar al tanto de dicha asunción, así que esto nunca debe hacerse. Y, si por cualquier motivo se hiciera, en ese caso, como mínimo debería emitirse un warning a la salida del compilador que informe del hecho de que se ha tomado un valor por defecto distinto del establecido inicialmente.

15.3.1.6.- Distribución de procesadores JSR 269 en ficheros .jar.

Como veremos más adelante, el procesamiento de anotaciones JSR 269 consta de varias fases, muy similares a las fases de todo desarrollo de software: análisis, diseño, implementación, anotación de código cliente y obtención de resultados.

Cuando se termina de desarrollar un procesador de anotaciones, puede que queramos distribuirlo a terceras partes para que puedan utilizarlo. En este subapartado vamos a describir las formas más habituales y las mejores prácticas a la hora de preparar la distribución de los procesadores de anotaciones que desarrollemos.

Lo primero a señalar es que **un procesador de anotaciones Java se distribuye de forma muy similar a cualquier otra aplicación Java: en uno o varios ficheros .jar (Java ARchive)**. Que sean uno o más ficheros .jar dependerá del mayor o menor nivel de fragmentación por el que opte el diseñador del procesador de anotaciones.

Los archivos .jar son ficheros comprimidos en formato ZIP, por lo que cualquier editor de ficheros ZIP, como el libre [7-Zip](#), permite visualizar sus contenidos. Dentro de los archivos .jar suelen encontrarse ficheros compilados de clase y, a veces, también los ficheros de código fuente correspondientes. Las aplicaciones Java ejecutables normalmente definen una clase como punto de entrada para ejecutar su método main en el “fichero de manifiesto”, situado en la ruta **META-INF/MANIFEST.MF**. En el caso de los procesadores esto no tiene ningún sentido, ya que **los procesadores de anotaciones no son aplicaciones Java al uso**.

Ficheros habituales en la distribución de procesadores de anotaciones		
Fichero	Ejemplo	Descripción
Tipo(s) anotación	<code>TipoAnotacion1.class</code> , <code>TipoAnotacion2.class</code>	Clases de los tipos anotación definidos. Es necesario que estén en las rutas de búsqueda de clases para que puedan ser reconocidos por el compilador y/o el entorno de ejecución.
Procesador(es)	<code>ProcesadorAnotacion1Processor.class</code> <code>ProcesadorAnotacion2Processor.class</code>	Clases de los procesadores de anotaciones definidos. Es necesario que estén en las rutas de búsqueda de procesadores para que <code>javac</code> los pueda descubrirlos e instanciarlos.
Fichero de descubrimiento de servicios (SPI)	Este es siempre el mismo y siempre está ubicado en la misma ruta canónica: META-INF/services/javax.annotation.processing.Processor	Fichero utilizado por <code>javac</code> para descubrir procesadores de anotaciones si no se utiliza su opción <code>-processor</code> para forzar la utilización de ciertos procesadores concretos.

Que sean uno o más ficheros .jar los utilizados para la distribución de los ficheros de nuestros procesadores de anotaciones se debe a que se puede optar por cualquiera de estas vías:

- (1) Colocar todos los ficheros en un único el fichero .jar por comodidad.
- (2) Separar los tipos anotación y los procesadores en dos o más ficheros.

Esta última opción es considerada más apropiada por muchos desarrolladores, ya que separa dos conceptos claramente distintos. En algunos casos, en función de cómo se quisieran estructurar las dependencias de código podrían ser necesarias sólo las definiciones de los tipos anotación, pero no los procesadores si no se va a realizar procesamiento de anotaciones. Esto permite mayor libertad a la hora de configurar las dependencias de una aplicación, además de que reduce el acoplamiento de los tipos anotación con sus procesadores, lo cual, como decimos, puede ser deseable en algunos casos y hace que algunos desarrolladores opten por esta opción.

El fichero `META-INF/services/javax.annotation.processing.Processor` sólo es necesario que se incluya en el fichero `.jar` donde se alojen los procesadores de anotaciones para que puedan ser descubiertos por `javac`. Puede no ser necesario si siempre se invocara `javac` con su opción `-processor`, que anula el proceso de descubrimiento de procesadores. Para más información sobre el proceso de descubrimiento de procesadores de anotaciones, puede consultarse el subapartado “Funcionamiento de `javac`”.

Supongamos que se opta por distribuir los ficheros de procesamiento de anotaciones del ejemplo de la tabla anterior en 2 ficheros `.jar`: uno para las definiciones de los tipos anotación y otro para las definiciones de los procesadores de anotaciones, con su respectivo fichero de descubrimiento ([SPI](#) = Service Provider Interface). La estructura de los ficheros `.jar` debería ser:

anotaciones-tipos.jar

```
nombre/canonico/paquete/anotacion1/TipoAnotacion1.class  
nombre/canonico/paquete/anotacion2/TipoAnotacion2.class
```

anotaciones-procesadores.jar

```
nombre/canonico/paquete/factorial/ProcesadorAnotacion1Processor.class  
nombre/canonico/paquete/factoria2/ProcesadorAnotacion1Processor.class  
META-INF/services/javax.annotation.processing.Processor
```

El fichero `META-INF/services/javax.annotation.processing.Processor` deberá contener una línea de texto por cada procesador de anotaciones. Cada línea será el nombre canónico de cada una de las clases procesador contenidas en el fichero `.jar`. En el fichero **anotaciones-procesadores.jar** tenemos 2 procesadores, por lo que su contenido será:

Fichero <code>META-INF/services/javax.annotation.processing.Processor</code>
<code>nombre.canonico.paquete.factorial.ProcesadorAnotacion1Processor</code>
<code>nombre.canonico.paquete.factoria2.ProcesadorAnotacion2Processor</code>

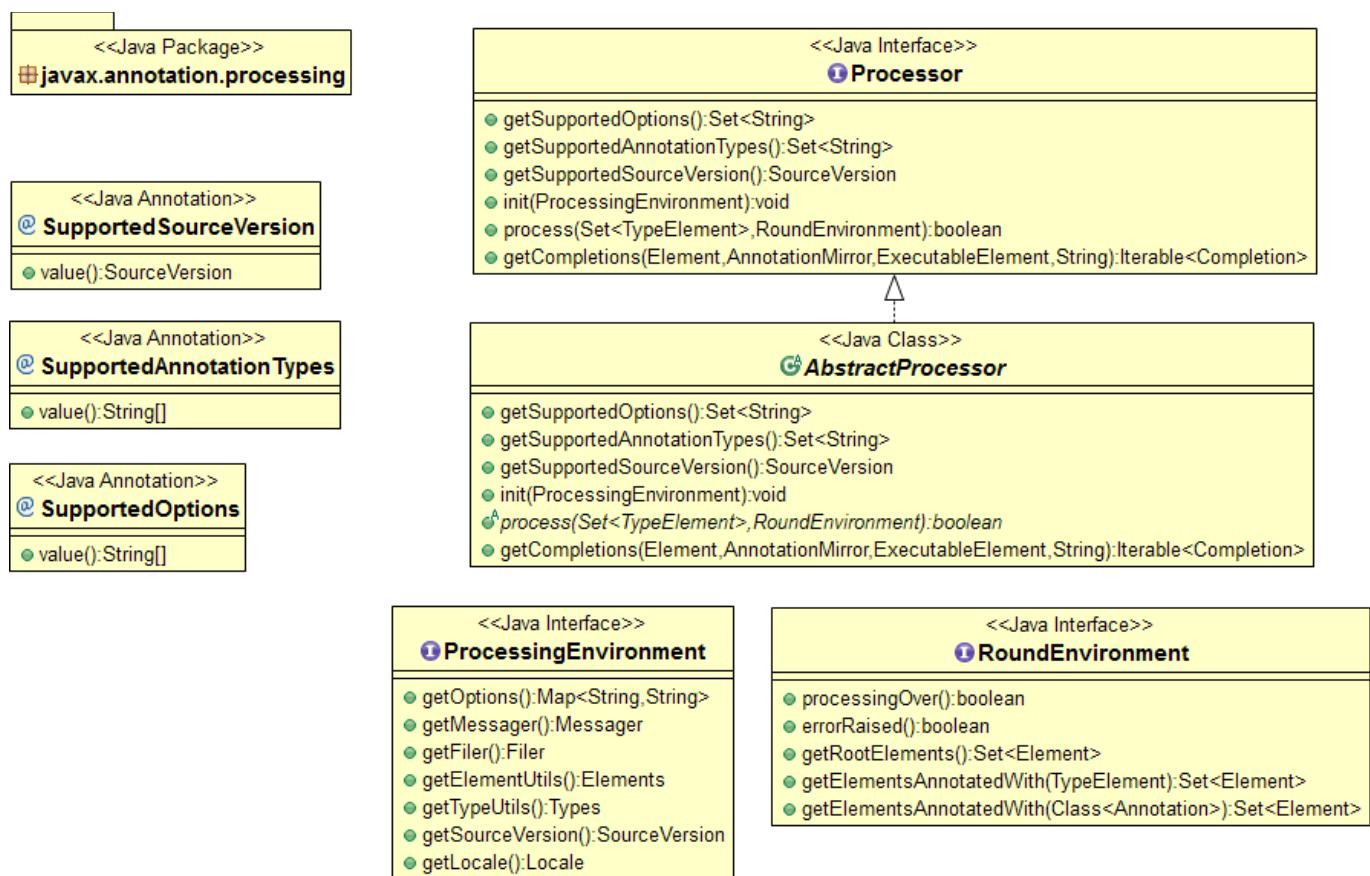
Si optamos por distribuir nuestros ficheros de procesamiento de anotaciones en un único fichero `.jar`, lo único que habrá que hacer es combinar la estructura descrita para ambos ficheros en uno solo.

NOTA: Los ficheros de procesamiento de anotaciones exportados en los ficheros `.jar` no tienen por qué estar necesariamente en dichos ficheros `.jar`. Pueden estar simplemente en el `CLASSPATH` o el `PROCESSORPATH`. Lo importante es que `javac` los descubra. No obstante, a la hora de distribuirlos a terceros, el procedimiento habitual es empaquetarlos en ficheros `.jar`.

Una vez que tengamos los ficheros de procesamiento de anotaciones bien empaquetados en nuestros ficheros `.jar` ya estaremos en disposición de distribuirlos a quienes queramos. No obstante, ya hemos dicho que los procesadores de anotaciones no son aplicaciones normales y no se ejecutan como tales (con el clásico `java -jar aplicacion.jar`), si no que se ejecutan a través de `javac`. Para saber cómo exactamente ha de realizarse la invocación a `javac` para iniciar el procesamiento, pueden consultarse los subapartados “Funcionamiento de `javac`” y “Opciones de linea de comandos de `javac`”.

15.3.2.- API de procesamiento: javax.annotation.processing.

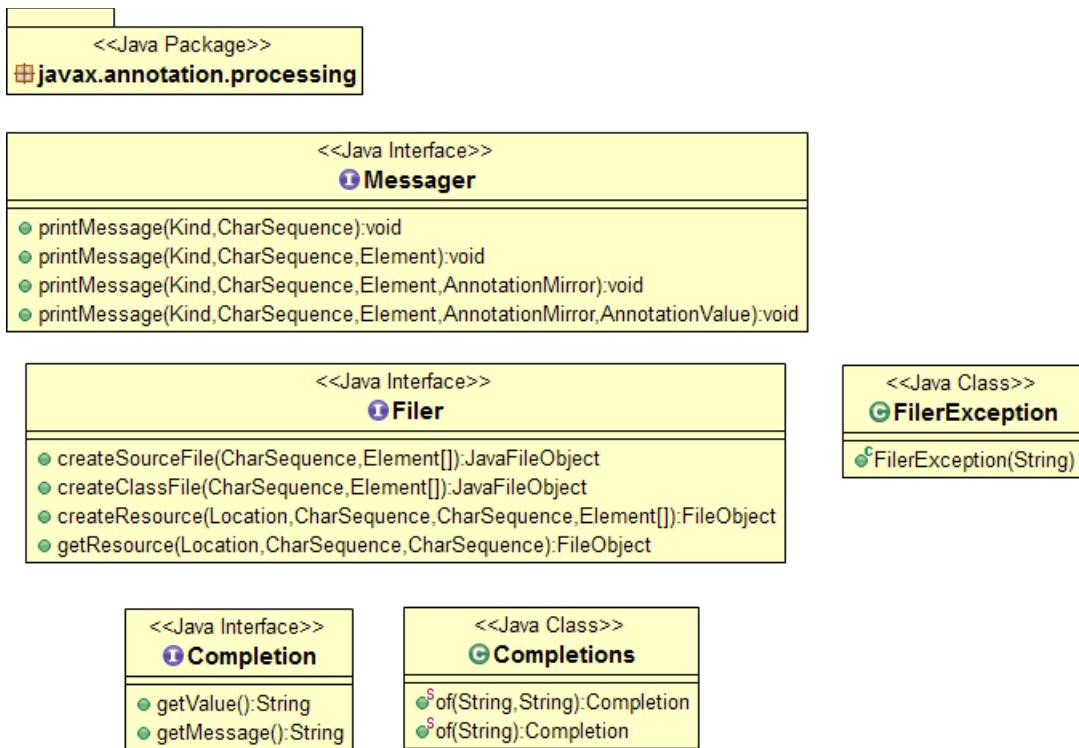
javax.annotation.processing – Procesadores de anotaciones	
Anotación / Interfaz / Clase	Descripción
<code>SupportedSourceVersion</code>	Anotación que indica el nivel del código fuente soportado por el procesador.
<code>SupportedAnnotationTypes</code>	Anotación que lista los tipos anotación soportados por el procesador.
<code>SupportedOptions</code>	Anotación que lista las opciones soportadas por el procesador.
<code>Processor</code>	Interfaz que deben implementar los procesadores de anotaciones JSR 269.
<code>AbstractProcessor</code>	Clase base abstracta que facilita la implementación de procesadores.
<code>ProcessingEnvironment</code>	Entorno de procesamiento. Proporciona información y clases de utilidad.
<code>RoundEnvironment</code>	Entorno de ronda. Información sobre la ronda y métodos de descubrimiento.



AbstractProcessor permite implementar un procesador de fácilmente heredando de ella e implementando su método `process`. Las anotaciones **SupportedX** permiten especificar las propiedades del procesador de anotaciones con anotaciones, con lo cual no habrá que implementar específicamente los métodos `getSupportedX` de **Processor**.

ProcessingEnvironment da acceso a información general del procesamiento, como las opciones de procesamiento pasadas al procesador, el nivel de código fuente o el **Locale** de internacionalización del entorno actual. No obstante, lo más importante es que proporciona instancias que implementan las facilidades básicas de procesamiento dadas por las interfaces **Messenger** y **Filer** (de esta misma API), y de **Elements** y **Types** (`javax.lang.model.util`).

javax.annotation.processing – Facilidades auxiliares	
Interfaz / Clase	Descripción
Messager	Permite emitir mensajes a la salida del compilador: información, warnings y errores.
Filer	Permite crear nuevos ficheros: de código fuente, de clase, binarios y de texto.
FilerException	Error al tratar de crear un fichero mediante Filer sin respetar sus restricciones.
Completion	“Terminación” de código sugerida por el procesador ante un texto de entrada incompleto.
Completions	Clase que permite construir instancias de Completion partiendo de un valor y/o mensaje.



Messager es la interfaz que implementa la comunicación con el compilador. Su método **printMessage** imprime mensajes a la salida del compilador que pueden ser de los tipos dados por **javax.tools.Diagnostic.Kind**: **NOTE**, **WARNING**, **MANDATORY_WARNING**, **ERROR** u **OTHER**. **Lo importante es que, al permitir emitir errores de compilación, esto habilita a los procesadores para implementar funciones de validación de código.**

Filer es la interfaz que habilita a los procesadores de anotaciones para la generación de nuevos ficheros. Es posible crear ficheros de tres tipos: (1) ficheros de código fuente, (2) ficheros de clase, y (3) ficheros de recurso, que podrán ser de texto o binarios. El método de los ficheros de recurso tiene más argumentos para poder especificar la ubicación (no una ubicación libre cualquiera, si no dentro de ciertas ubicaciones estándar), el paquete al que mapearlos (si es que así se desea), y el nombre completo del fichero (incluida extensión).

Completion es la interfaz que modela las “terminaciones” de código. Una “terminación” (traducción del original “completion”) es una sugerencia de cómo terminar un fragmento de código. Esta es una **facilidad añadida a los procesadores de anotaciones para una mejor integración en tiempo real con los IDEs**. La idea es que el usuario empiece a escribir parte de un texto y el procesador, a través de su método **getCompletions** devuelva sugerencias de posibles terminaciones, acompañadas opcionalmente de mensajes.

15.3.3.- Language Model API: javax.lang.model.

El procesamiento de anotaciones JSR 269, al tener que acceder a la información de los elementos de código fuente del lenguaje Java para poder procesarlos, necesita de una API que modele de forma abstracta las construcciones del lenguaje.

El procesamiento de anotaciones J2SE 1.5 disponía de la Mirror API para tal propósito y, con motivo de la introducción de la especificación de la JSR 269, se aprovechó para crear una nueva API de modelado del lenguaje Java, fuertemente basada en la Mirror API, pero remozada y mejorada gracias a toda la experiencia acumulada anterior.

Esta **API de modelado del lenguaje Java**, que podríamos llamar **Language Model API**, aunque no es realmente el nombre con el que se la denomina habitualmente, se corresponde con los paquetes y subpaquetes contenidos en **javax.lang.model**.

Paquetes de la Language Model API (javax.lang.model)	
Paquete	Descripción
javax.lang.model.element	Modelado del lenguaje Java. Declaraciones de elementos de código fuente.
javax.lang.model.type	Modelado del lenguaje Java. Tipos simbólicos.
javax.lang.model.util	Clases de utilidad para el procesamiento. Clases e interfaces Visitor.

La **función de la Language Model API** es proporcionar un modelo de representación lo más completo posible de los elementos de código fuente del lenguaje de programación Java. Estas representaciones abstractas vendrán modeladas por una serie de interfaces que expondrán una gran diversidad de métodos para poder acceder a la información dada por lo que sería una especie de **imagen del código fuente en tiempo de compilación**.

La **Language Model API** proporciona una visión de la estructura de un programa Java en tiempo de compilación, basada en el código fuente escrito y con un acceso de sólo lectura. El proceso es el siguiente: (1) se lee el fichero de código fuente `.java`, (2) el compilador analiza la estructura del código y crea un meta-modelo del mismo en memoria, (3) se crea a partir del meta-modelo la información necesaria sobre la estructura del código analizado y se hace disponible para su publicación a través de la Model API.

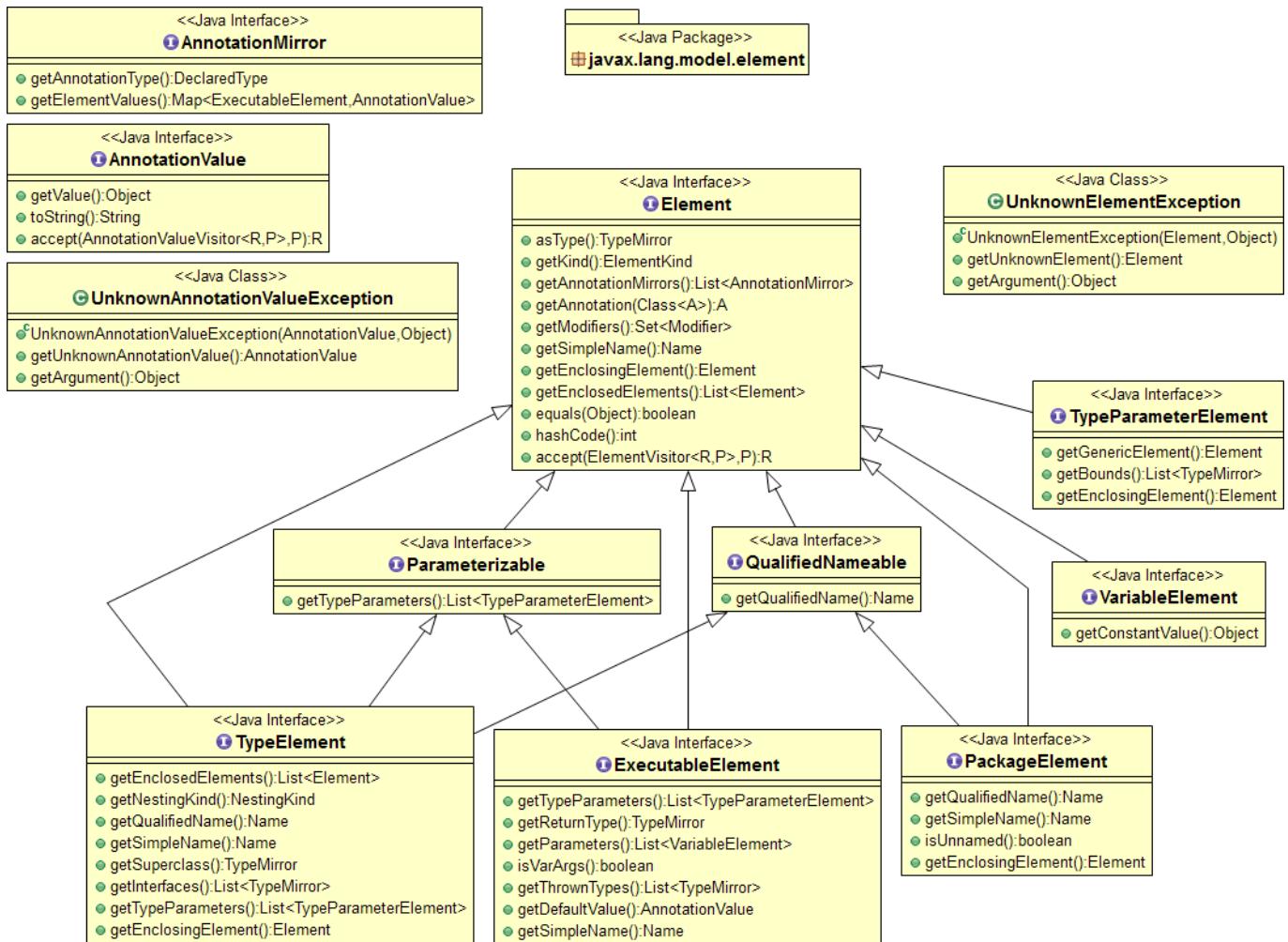
La **Language Model API** por tanto modela las construcciones semánticas de un programa escrito en Java tomando como referencia los fragmentos de código fuente de las entidades de dicho programa. Proporciona interfaces que representan todas las entidades que se pueden declarar en un programa Java, tales como paquetes, clases, métodos, campos, etcétera.

La información proporcionada por la Language Model API será utilizada por los procesadores de anotaciones para implementar su lógica de procesamiento. No obstante, a diferencia de la Mirror API que estaba totalmente enfocada al procesamiento de anotaciones, la Language Model API se concibe como una API de propósito general de modelado del lenguaje Java para tareas que incluyen (pero no se limitan) al procesamiento de anotaciones.

IMPORTANTE: La Language Model API no modela los cuerpos de los métodos, por lo que a través de la misma no podremos obtener información de las propiedades de los elementos del lenguaje que se encuentren dentro del cuerpo de un método. Para obtener esa información **habremos de usar la Compiler Tree API**. Para más detalles, véase el procesador de anotaciones ejemplo del tipo anotación `@ImprimirMensaje para Java SE 6/7`.

15.3.3.1.- Paquete javax.lang.model.element.

javax.lang.model.element – Elementos	
Interfaz / Clase	Descripción
AnnotationMirror	Representa el elemento de una anotación y guarda sus valores.
AnnotationValue	Valor de un elemento dado de una anotación.
UnknownAnnotationValueException	Arrojada por un Visitor ante un tipo de valor de anotación desconocido.
Element y sus subinterfaces	Interfaces que modelan los elementos de código del lenguaje Java.
UnknownElementException	Excepción arrojada por un Visitor ante un tipo de elemento desconocido.

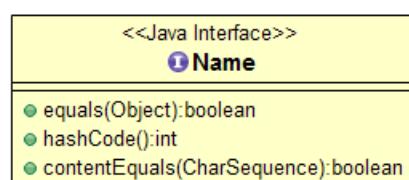
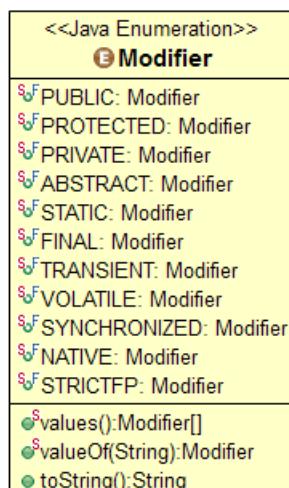
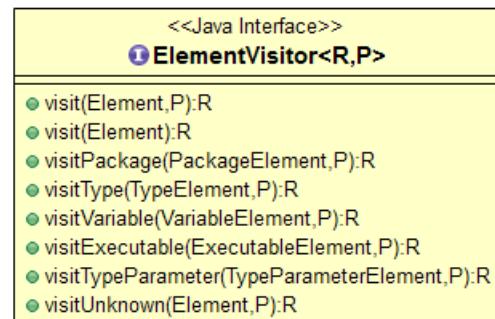
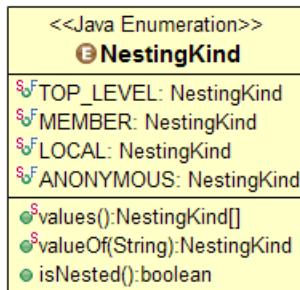
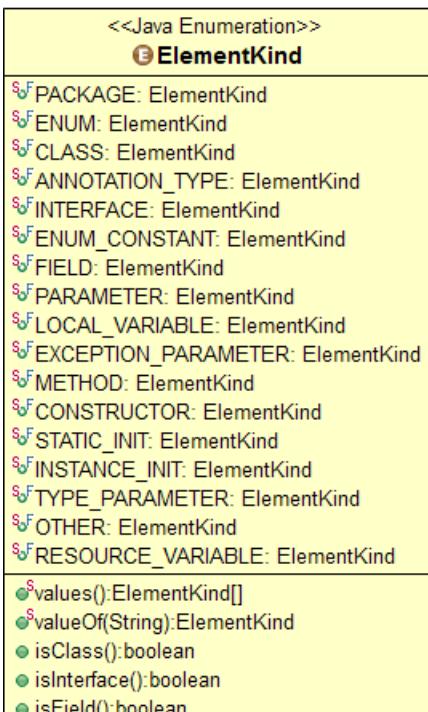
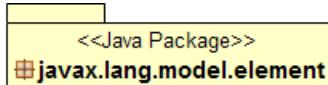


AnnotationMirror es vital, ya que será la interfaz de la que extraeremos información sobre los valores de los elementos de las anotaciones concretas del código fuente a procesar. **CUIDADO:** **getElementValues** sólo devuelve los valores de los elementos directamente presentes en el código. Para recuperarlos todos, los presentes y los no presentes con sus valores por defecto, se puede utilizar **Elements.getElementValuesWithDefaults** de **util**.

La interfaz **Element** es la base de la jerarquía de todos los elementos de programa Java. Se ha simplificado la jerarquía con respecto al paquete **com.sun.mirror.declaration** pasando todos los diferentes tipos de elementos a un tipo enumerado llamado **ElementKind** (ver página siguiente). Por lo demás, ambas excepciones definidas se refieren a casos en los que un Visitor se encontrara con algún tipo nuevo de elemento debido a una evolución del lenguaje.

javax.lang.model.element – Clases de elementos, modificadores y Visitors

Interfaz / Clase	Descripción
<code>ElementKind</code>	Enumerado que lista todas las clases de tipos de elementos del lenguaje Java.
<code>NestedKind</code>	Enumerado que lista todos los posibles niveles de anidamientos de un elemento.
<code>Modifier</code>	Enumerado que lista todos los posibles modificadores del lenguaje Java.
<code>Name</code>	Interfaz para los nombres de elementos que garantiza una correcta comparación.
<code>ElementVisitor</code>	Interfaz para visitar los distintos tipos de elementos del lenguaje Java.
<code>AnnotationValueVisitor</code>	Interfaz para visitar los valores de los elementos de las anotaciones según su tipo.

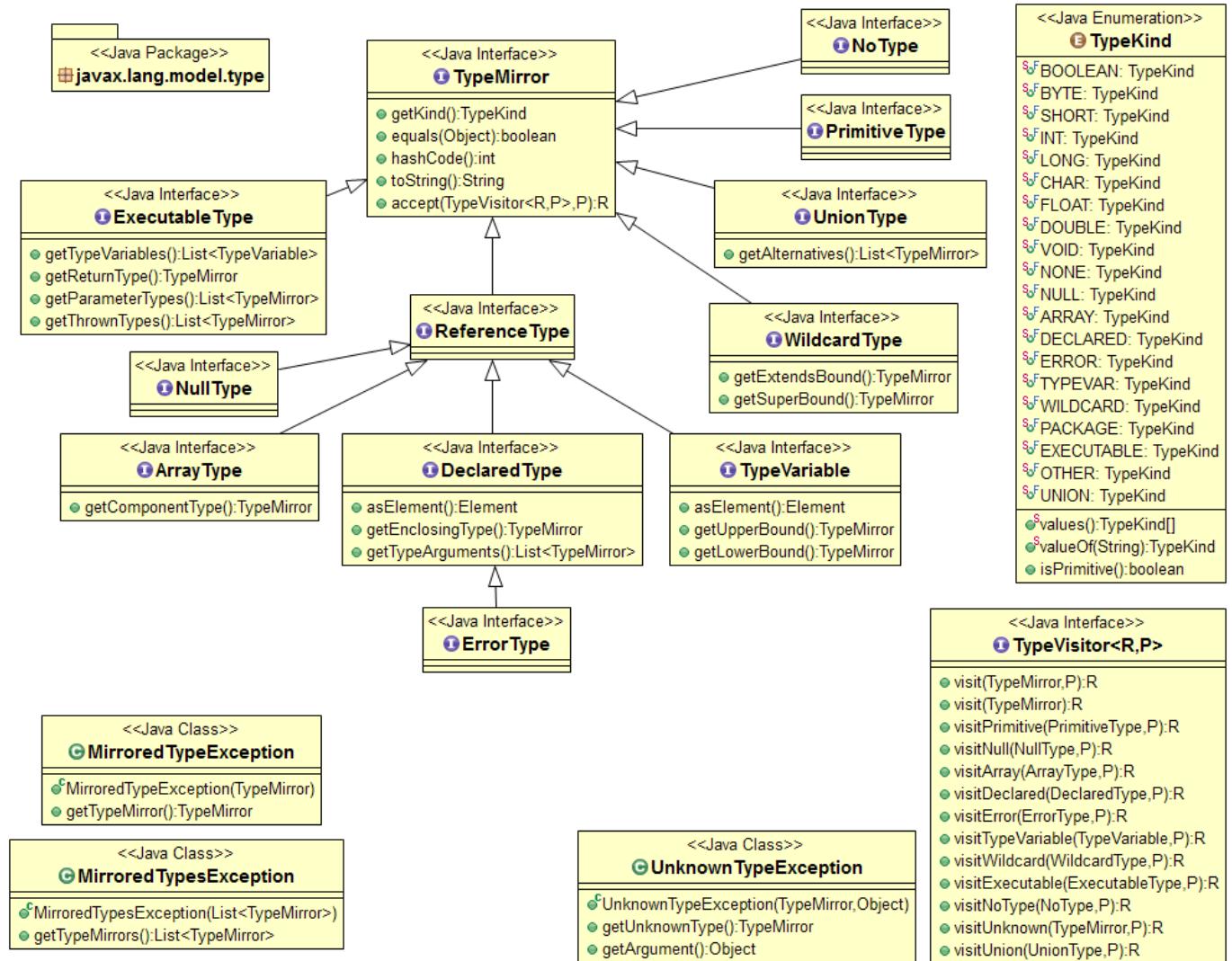


ElementKind modela todos los tipos de elementos del lenguaje Java y **NestingKind** su tipo de anidación. **Modifier** hace lo propio con los modificadores que se pueden establecer sobre elementos del lenguaje. Todos estos son enumerados porque se ha tenido muy en cuenta que Java es un lenguaje que evoluciona y podría incluir en el futuro nuevos elementos dentro de estas categorías, siendo sencillo en tal caso simplemente nuevos valores a dichos enumerados.

ElementVisitor y la curiosa **AnnotationValueVisitor** son las interfaces Visitor que permiten navegar por los elementos y tipos de valores de las anotaciones respectivamente. **Name** simplemente encapsula los nombres de los elementos en una clase que garantiza la correcta comparación de nombres a través de una adecuada implementación del contrato del método `equals`, ofreciendo además `contentEquals` para comparación directa de caracteres.

15.3.3.2.- Paquete javax.lang.model.type.

javax.lang.model.type – Tipos, Clases de tipos y Visitores	
Interfaz / Clase	Descripción
TypeMirror y sus subinterfaces	Interfaces que modelan todos los tipos del lenguaje Java.
TypeKind	Enumerado que lista todos los posibles de un elemento del lenguaje Java.
MirroredType[s]Exception	Excepciones producidas al intentar acceder a la Class de un TypeMirror.
TypeVisitor	Interfaz para visitar los distintos tipos del lenguaje Java.
UnknownTypeException	Excepción arrojada por un Visitor ante un tipo desconocido.



La interfaz **TypeMirror** es la base de la jerarquía de los tipos Java. **ReferenceType** agrupa todos los tipos por referencia. **TypeVariable** modela la genericidad y **WildcardType** los comodines (?) y sus límites (“bounds”) superior (? extends T) e inferior (? super T). **ErrorType** modela los tipos que no se han podido cargar correctamente. **NOTA:** **UnionType** es un tipo nuevo de Java SE 7 que modela la unión de tipos excepción en los [try multi-catch](#).

TypeKind modela todos los tipos en un enumerado por si vinieran nuevos tipos en el futuro. **TypeVisitor** será la interfaz que nos permita navegar con Visitors a través de los diferentes tipos de los elementos de nuestro código. Finalmente, se incluyen las excepciones de acceso a los elementos a la propiedad **Class** de los Mirrors (esto está explicado más adelante).

15.3.3.3.- Paquete javax.lang.model.util.

javax.lang.util – Facilidades sobre Elementos y Tipos	
Interfaz / Clase	Descripción
ElementFilter	Clase con métodos de conveniencia para el filtrado de elementos.
Elements	Interfaz con métodos de conveniencia para ciertas operaciones con elementos.
Types	Interfaz con métodos de conveniencia para ciertas operaciones con elementos.

<pre><<Java Package>> javax.lang.model.util</pre>	ElementFilter <pre> fieldsIn(Iterable<Element>):List<VariableElement> fieldsIn(Set<Element>):Set<VariableElement> constructorsIn(Iterable<Element>):List<ExecutableElement> constructorsIn(Set<Element>):Set<ExecutableElement> methodsIn(Iterable<Element>):List<ExecutableElement> methodsIn(Set<Element>):Set<ExecutableElement> typesIn(Iterable<Element>):List<TypeElement> typesIn(Set<Element>):Set<TypeElement> packagesIn(Iterable<Element>):List<PackageElement> packagesIn(Set<Element>):Set<PackageElement></pre>	Types <pre> asElement(TypeMirror):Element isSameType(TypeMirror,TypeMirror):boolean isSubtype(TypeMirror,TypeMirror):boolean isAssignable(TypeMirror,TypeMirror):boolean contains(TypeMirror,TypeMirror):boolean isSubsignature(ExecutableType,ExecutableType):boolean directSupertypes(TypeMirror):List<TypeMirror> erasure(TypeMirror):TypeMirror boxedClass(PrimitiveType):TypeElement unboxedType(TypeMirror):PrimitiveType capture(TypeMirror):TypeMirror getPrimitiveType(TypeKind):PrimitiveType getNullType():NullType getNoType(TypeKind):NoType getArrayType(TypeMirror):ArrayType getWildcardType(TypeMirror,TypeMirror):WildcardType getDeclaredType(TypeElement,TypeMirror[]):DeclaredType getDeclaredType(DeclaredType,TypeElement,TypeMirror[]):DeclaredType asMemberOf(DeclaredType,Element):TypeMirror</pre>
Elements <pre> getPackageName(CharSequence):PackageElement getTypeElement(CharSequence):TypeElement getElementValuesWithDefaults(AnnotationMirror):Map<ExecutableElement,AnnotationValue> getDocComment(Element):String isDeprecated(Element):boolean getBinaryName(TypeElement):Name getPackageName(Element):PackageElement getAllMembers(TypeElement):List<Element> getAllAnnotationMirrors(Element):List<AnnotationMirror> hides(Element,Element):boolean overrides(ExecutableElement,ExecutableElement,TypeElement):boolean getConstantExpression(Object):String printElements(Writer,Element[]):void getName(CharSequence):Name</pre>		

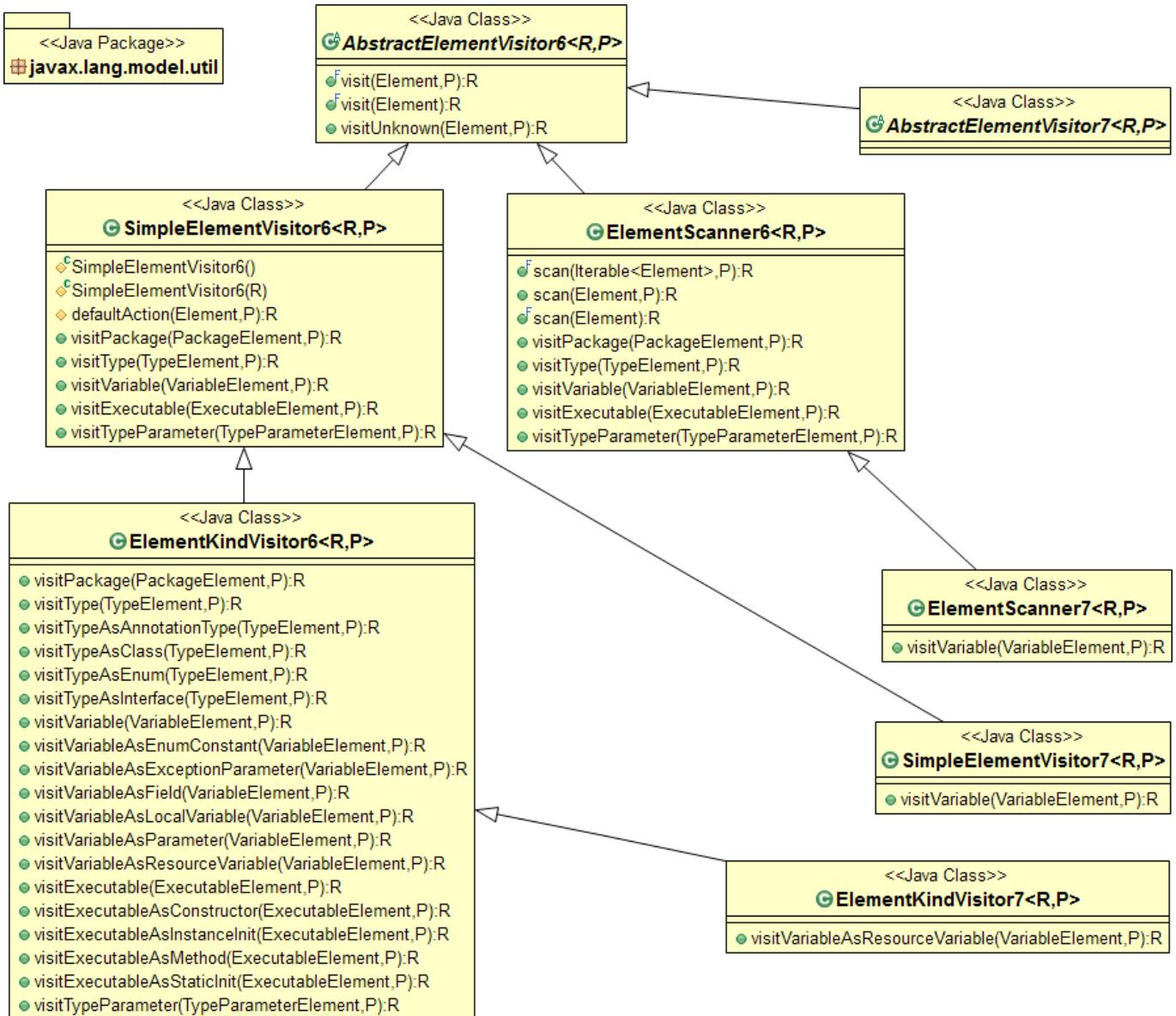
ElementFilter nos permite filtrar de forma cómoda por tipos de elementos (paquetes, tipos, métodos, constructores y campos) dentro de una colección iterable de elementos. No obstante, hay que comentar que esta clase nos parece un paso atrás, ya que no tiene la potencia de los filtros que podríamos construir usando la clase **DeclarationFilter** de la Mirror API (véase el apartado “Filtrado de declaraciones” del procesamiento J2SE 1.5). Debido a esto, cuando necesitemos filtrar por un criterio más exigente que simplemente el tipo de elemento, tendremos que montar un bucle **for** o, llamar a un método auxiliar si hemos refactorizado el código de búsqueda precisamente para no abusar de las construcciones de bucles iterativos.

Elements incluye métodos de conveniencia de elementos. Especialmente interesantes son los que recuperan información sobre las anotaciones: **getElementValuesWithDefaults** (obtiene todos sus valores, incluyendo los valores por defecto) y **getAllAnnotationMirrors** (para obtener todas las anotaciones sobre un elemento, incluyendo la herencia).

Types incluye todos los métodos de conveniencia para tipos. Importantes son los que permiten comparación de tipos, como **isSameType** e **isAssignable**, ya que no se debe utilizar **equals** para comparar tipos: **t1.equals(t2)** y **Types.isSameType(t1, t2)** darán resultados diferentes. Esto es así porque no hay ninguna garantía de que un tipo sea representado siempre por el mismo objeto particular. De forma similar, tampoco se debe utilizar **instanceof** para preguntar por el tipo de elemento, si no utilizar **getKind()**.

javax.lang.model.util – Visitors de Elementos

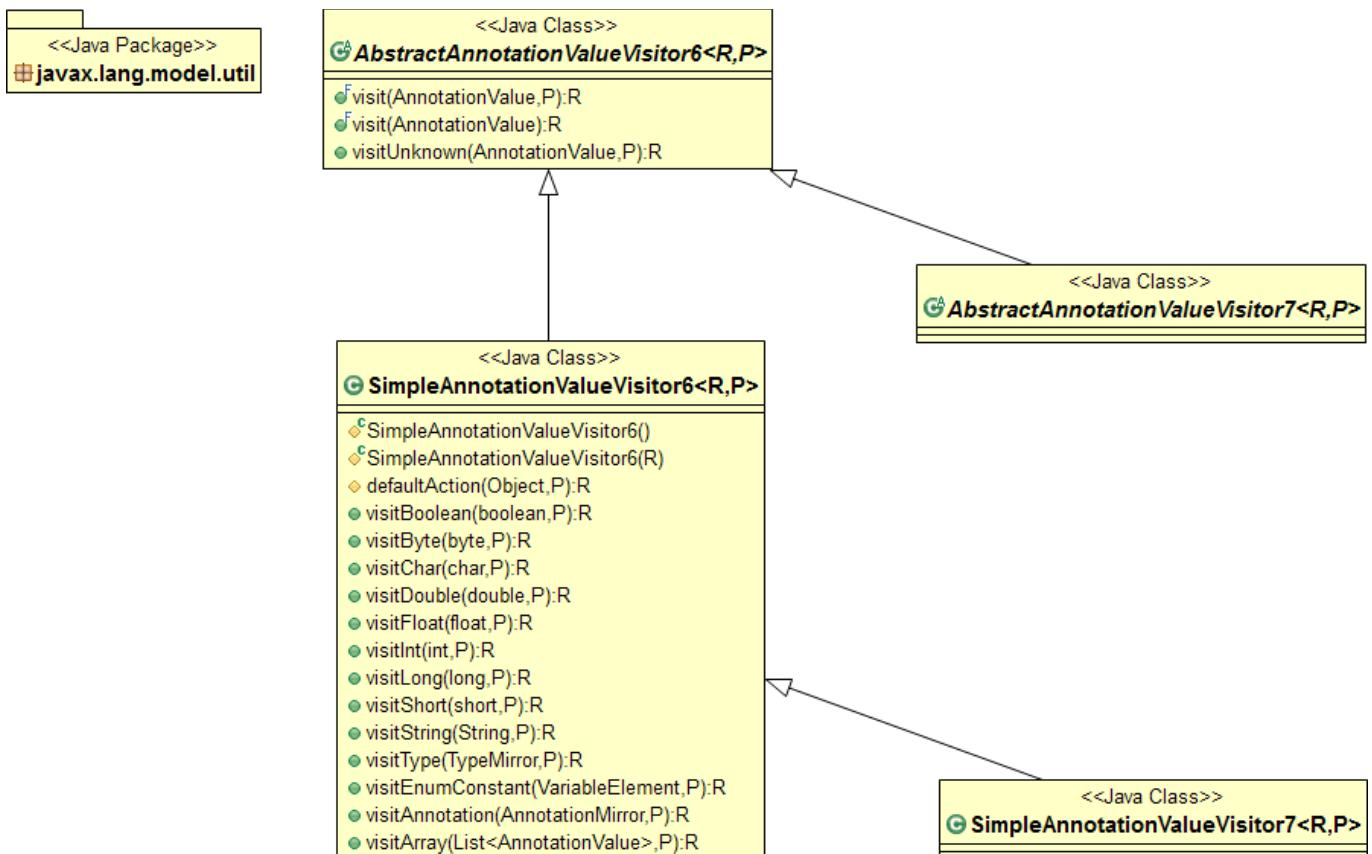
Interfaz / Clase	Descripción
<code>AbstractElementVisitor[6/7]</code>	Clase abstracta sobre la que implementar Visitor de elementos Java SE 6/7.
<code>SimpleElementVisitor[6/7]</code>	Visitor sencillo que llama a <code>defaultAction</code> sobre elementos Java SE 6/7.
<code>ElementKindVisitor[6/7]</code>	Visitor según los distintos <code>ElementKind</code> de los elementos Java SE 6/7.
<code>ElementScanner[6/7]</code>	Visitor que visita con <code>scan</code> los subelementos de elementos Java SE 6/7.



Tenemos un buen surtido de Visitors sobre elementos: la clase abstracta raíz (por si queremos implementar un Visitor propio), implementación sencilla, según su `ElementKind` y finalmente el scanner (o “rastreador”), que visitará elementos y subelementos.

NOTA: Los Visitor de elementos Java SE 7 son iguales a Java SE 6, salvo por la particularidad de que están preparados para reconocer el nuevo `ElementKind.RESOURCE_VARIABLE`, que es el tipo de elemento que modela las variables de recursos de los nuevos bloques `try-with-resources` introducidos en Java SE 7. El método `visitVariableAsResource` es, por tanto, nuevo de Java SE 7 y los Visitor6, que ahora incluyen este método, lo implementan llamando a `visitUnknown`.

javax.lang.model.util – Visitors de Valores de anotación	
Interfaz / Clase	Descripción
AbstractAnnotationValueVisitor[6/7]	Clase base abstracta sobre la que implementar los Visitor según los tipos de valores posibles de una anotación Java SE 6/7.
SimpleAnnotationValueVisitor[6/7]	Implementación sencilla del Visitor que llama a defaultAction para implementar una acción por defecto según los tipos de valores posibles de una anotación Java SE 6/7.

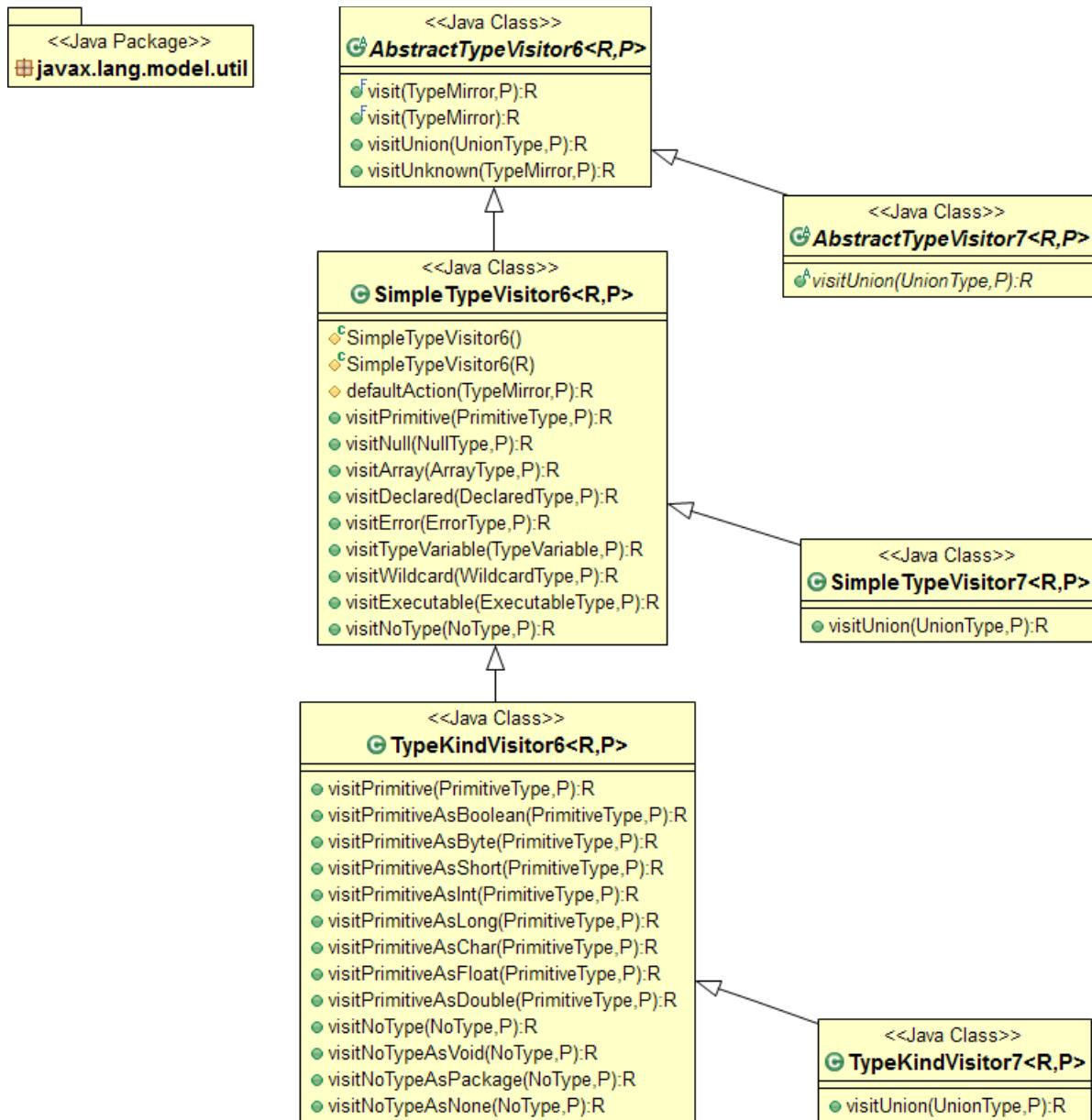


La API de procesamiento JSR 269 incluye un nuevo tipo de Visitors, que no se incluían en la anterior Mirror API: los **AnnotationValueVisitor**, y que permiten visitar los valores de las anotaciones (instancias de la interfaz **AnnotationValue**) que hayamos podido descubrir y recopilar en alguna colección y que necesitemos procesar, según los distintos tipos de retorno de los valores de sus elementos, disponiendo de un método **visitX** por cada tipo posible.

Los valores de elementos de un tipo anotación no pueden ser de cualquier tipo, si no que se les imponen varias restricciones. Como ya explicamos cuando describimos las características de los tipos anotación, en el subapartado “Tipos permitidos para los elementos”, los tipos que pueden tener los elementos de un tipo anotación son exclusivamente los siguientes:

- Tipos primitivos (boolean, byte, int, char, short, long, float y double).
- String.
- Enumerados.
- **Class**, tal cual o con parámetros de tipo (**Class<T>**, **Class<? extends Collection>**).
- Otros Tipos Anotación previamente definidos.
- Un array de cualquiera de los tipos anteriores.

javax.lang.model.util – Visitors de Tipos	
Interfaz / Clase	Descripción
<code>AbstractTypeVisitor[6/7]</code>	Clase abstracta sobre la que implementar Visitor de tipos Java SE 6/7.
<code>SimpleTypeVisitor[6/7]</code>	Visitor sencillo que llama a <code>defaultAction</code> sobre los tipos Java SE 6/7.
<code>TypeKindVisitor[6/7]</code>	Visitor según los distintos <code>TypeKind</code> de los tipos Java SE 6/7.



Para navegar según los tipos tenemos una jerarquía idéntica a la de los Visitor de los elementos, salvo por la ausencia de los scanner, que no tienen sentido en los tipos. Tenemos la clase abstracta raíz (para facilitarnos el implementar nuestro propio `TypeVisitor`), una implementación sencilla, y finalmente el Visitor según la clase de tipo dada por `TypeKind`.

Como comentamos en el paquete de los tipos, en Java SE 7 se añadió el tipo `UnionType` para modelar la unión de tipos excepción en los [try multi-catch](#). Precisamente la principal diferencia entre los Visitor6 y los Visitor7 es que los nuevos incorporan la posibilidad de visitar este nuevo tipo con el método `visitUnion`, mientras que la implementación de los Visitor6 cuando se llame a `visitUnion`, al no existir tal tipo en Java SE 6, delegarán en `visitUnknown`.

15.3.3.4.- Elementos vs Tipos.

Al igual que en el procesamiento J2SE 1.5 era necesario tener muy clara la distinción entre Declaraciones y Tipos (para más detalles, puede consultar el subapartado “Declaraciones vs Tipos”), dicha distinción también existe de forma ligeramente diferente en la API de procesamiento de anotaciones JSR 269, ya que esta distinción es provocada por la existencia de la genericidad en el lenguaje Java y seguirá existiendo mientras Java tenga genericidad.

Así pues, antes de empezar a escribir código usando la API de modelado del lenguaje Java o Language Model API es necesario distinguir con total claridad dos conceptos distintos que son muy importantes: la **distinción entre Elementos y Tipos**. De hecho, dado que se sabe que esto provoca cierta confusión entre los usuarios de la Language Model API, en la documentación oficial de la API se hace mucho hincapié en tratar de explicar esta separación de conceptos.

Los Elementos (paquete `javax.lang.model.element`), **son abstracciones de fragmentos del código fuente que representan un elemento de código (de ahí su nombre) del lenguaje Java**. Es muy importante que tengamos claro que un elemento siempre representa un determinado fragmento del código fuente. Es decir, que **los elementos se corresponden uno-a-uno con fragmentos de código fuente**. Por ejemplo: al declarar una clase `public class Clase<T>`, esto se representará como `TypeElement`, ya que es un elemento de código fuente que define un tipo.

Recordemos que los elementos Java, según vimos en el diagrama de clases de su paquete, se pueden clasificar de entrada en las siguientes interfaces: `PackageElement`, `TypeElement`, `ExecutableElement`, `VariableElement` y `TypeParameterElement`. La distinción, dentro de estas interfaces, de las clases de elementos concretos se realizará a través del método `getKind()` de la instancia concreta del elemento, que nos devolverá el valor del enumerado `ElementKind` que nos identificará exactamente con qué clase de elemento estamos tratando.

Los Tipos (paquete `javax.lang.model.type`) **son abstracciones de los tipos de datos Java una vez son resueltos por el compilador para un uso específico**. La necesidad de incluir este tipo de entidades se debe a que sirven para modelar la genericidad.

Correspondencia Elemento de código fuente ↔ Tipos posibles		
	Declaración de elemento de código fuente (parámetros formales)	Tipos posibles (parámetros reales)
Sin genericidad	<code>java.util.List</code>	<code>java.util.List</code>
Con genericidad	<code>java.util.List<T></code>	<code>java.util.List<T> (IMPOSIBLE)</code> <code>java.util.List<String></code> <code>java.util.List<Integer></code> ... <code>java.util.List</code>
	<code>java.util.List<String></code>	<code>java.util.List<String></code>
	<code>java.util.List<Integer></code>	<code>java.util.List<Integer></code>

	<code>java.util.List</code>	<code>java.util.List</code>

NOTA: Los Elementos representan código fuente, como `TypeElement`, que representa elementos de tipo en el código, como clases o interfaces. No obstante, `TypeElement` no contiene información sobre la clase en sí misma. `TypeElement` contiene el nombre de la clase, pero no información acerca de ella, como quién es su superclase. Ese tipo de información está accesible sólo a través de un `TypeMirror`. Puede accederse al `TypeMirror` de un `Element` mediante `element.asType()`.

La declaración `public class Clase<T>`, era de tipo `Clase<T>`, ya que los elementos siempre tienen como tipo el que se haya escrito directamente en el código fuente. Mientras tanto, su objeto de tipo, que en este caso sería un `DeclaredType`, podría corresponder tanto a `Clase<String>`, como a `Clase<Integer>`, o incluso simplemente a `Clase` (un “raw type”, traducible como “tipo basto”, “tipo raso” o “tipo simple”). Así pues, **un elemento puede producir múltiples tipos si tiene un parámetro de tipo T**. No obstante, un elemento sin tipo parámetro como `Clase<Boolean>` sólo producirá el mismo tipo `Clase<Boolean>`.

La clave para diferenciar entre elemento y tipo es darse cuenta de que **un tipo nunca puede estar indeterminado**. Es decir: un elemento puede ser `Clase<T>` si así se declara en el código fuente, pero un tipo no. Nótese como en la tabla ilustrativa el tipo genérico no concretado `java.util.List<T>` aparece ~~tachado~~. Esto es así para indicar claramente que eso es imposible. Y es que, cuando el código es analizado por el compilador, `T` será asignado a un tipo concreto, o de la estructura del código se deberá poder inferir que así será en su ejecución.

Debido a lo anterior, a los parámetros de tipo de instancias de elementos se les conoce como **“parámetros formales” (“formal parameters”)** y a los parámetros de tipo de las instancias de tipos se les conoce como **“parámetros reales” (“actual parameters”)**.

Los **“parámetros formales”** son los **parámetros de tipo directamente escritos en el código fuente por el programador**. Deben su nombre de “formales” al hecho de que sirven para realizar una especificación de la forma que tiene que tener un tipo sin llegar a concretarlo necesariamente. En Java, los parámetros formales se implementan a través de los parámetros genéricos de tipo como `<T>`, así como los tipos comodín (“wildcard”), que no son más que parámetros formales a los que se les añade la restricción de unos límites (“bounds”) superior (`<? extends T>`) e inferior (`<? super T>`).

Los **“parámetros reales”** son los que se resuelven utilizar realmente cuando se ejecuta el código (de ahí lo de “reales”). En Java, los parámetros reales son inferidos por el compilador mediante sus reglas de resolución o “inferencia de tipos”, de tal manera que de un elemento con parámetros formales siempre se puedan extraer los parámetros reales.

Los parámetros formales corresponden a los dados por los elementos, y no en todos los tipos de elementos, ya que **hay elementos a los que no les corresponde ningún tipo**, como los elementos de paquetes, a los que no tendrá sentido tratar de asignarles un tipo. De hecho, existe el tipo `NoType`, un pseudo-tipo especial que modela la no aplicabilidad de un tipo a un elemento. Los valores de `TypeKind` aplicables a este pseudo-tipo son los siguientes:

Valores del enumerado <code>TypeKind</code> con sentido para el tipo <code>NoType</code>	
<code>TypeKind</code>	Descripción
<code>VOID</code>	Corresponde a la palabra clave <code>void</code> , aplicable en elementos ejecutables <code>ExecutableElement</code> (es decir: en métodos, constructores e inicializadores de instancia y estáticos).
<code>PACKAGE</code>	El pseudo-tipo que representa un “no-tipo” para un <code>PackageElement</code> .
<code>NONE</code>	Para otros casos donde no tenga sentido un tipo, como la superclase de <code>Object</code> .

Ejemplo: Diferencias entre Elementos y Tipos en las definiciones de clase

Imaginemos que definimos las clases siguientes:

```
public class ClaseA<T>
public class ClaseB<T extends Number> extends ClaseA<T>
public class ClaseC extends ClaseB<Integer>
```

La **ClaseA** tiene un parámetro de tipo, la **ClaseB** hereda de **ClaseA** y le coloca un límite superior a la definición del tipo, forzando a que sea o herede de la clase **Number**. Finalmente, la **ClaseC** simplemente concretiza el parámetro de tipo a **Integer** (que efectivamente hereda de **Number**) y hereda de dicha clase para crear una clase no parametrizada al final de la jerarquía. Veamos qué valores de elementos y tipos tenemos para cada clase. Además del tipo de la clase (obtenido con `Types.getDeclaredType(elementoClase)`), para aclarar otro concepto que explicaremos a continuación incluimos también el valor del “tipo de la superclase” o “supertipo” (obtenido con `Types.directSupertypes(claseDeclaredType)`).

Correspondencia Declaración en código fuente ↔ Tipos posibles			
Clase	Elemento (parámetros formales)	Tipo (parámetros reales)	SuperTipo o Tipo de la superclase (parámetros reales)
ClaseA	ClaseA<T>	ClaseA	java.lang.Object
ClaseB	ClaseB<T extends java.lang.Number>	ClaseB	ClaseA
ClaseC	ClaseC	ClaseC	ClaseB<java.lang.Integer>

¿Cómo interpretamos estos resultados? Lo primero es ver cómo se pierde la información de los parámetros de tipo al pasar de Elemento a Tipo. ¿Pero eso quiere decir que un tipo nunca va a tener información de parámetros de tipo? No, como demuestra el tipo de la superclase de **ClaseC**, que es **ClaseB<java.lang.Integer>**.

Entonces, ¿por qué el tipo de una declaración nunca tiene parámetros de tipo en su propio tipo? Pues porque el parámetro de tipo sólo se puede concretizar en las subclases. **ClaseA** dentro de su definición nunca puede darle un valor a **T**, por lo que el compilador le asigna como tipo su tipo simple. Sólo cuando hay herencia se le puede dar un valor a **T**. En el caso de **ClaseC**, como se ve en su declaración, hereda de **ClaseB<java.lang.Integer>**, que es justo lo que la Model API devuelve como tipo de su superclase. En este caso, **T** ha sido concretado como **Integer**, respetando además la restricción dada por **ClaseB** de que tenía que ser un **Number**.

De este ejemplo podemos extraer una noción muy importante para poder separar y discernir correctamente los conceptos de declaración y de tipo: todos los elementos tienen asociados un objeto **Element** y un **TypeMirror**, pero, además, si el elemento define un parámetro de tipo variable **<T>**, cada subclase puede tener un supertipo específico. Este es el caso de **ClaseC** con **ClaseB<java.lang.Integer>**. Si creáramos una **ClaseC2** que también heredara de **ClaseB** con `public class ClaseC2 extends ClaseB<Float>`, tendríamos ese resultado del que tanto se habla en la documentación oficial: **ClaseB<T extends java.lang.Number>** no produce únicamente un tipo (antigua correspondencia uno-a-uno, cuando no había genericidad), si no múltiples tipos concretos: **ClaseB<java.lang.Integer>** y **ClaseB<java.lang.Float>**, en el caso de este ejemplo.

15.3.3.5.- Mirrors: A través del espejo.

La Language Model API hereda de la Mirror API los “Mirrors”. De hecho, este apartado es casi idéntico al dedicado a los “Mirrors” de la Mirror API en el apartado de procesamiento J2SE 1.5. Aunque son muy importantes, los “Mirrors” con pocos; básicamente dos interfaces, una para modelar las anotaciones y otra para los tipos, junto con dos excepciones relacionadas:

Mirrors de la Language Model API y sus excepciones	
Nombre cualificado	Descripción
<code>javax.lang.model.element.AnnotationMirror</code>	Representa el elemento de una anotación.
<code>javax.lang.model.type.TypeMirror</code>	Base de la jerarquía de tipos de Language Model API.
<code>javax.lang.model.type.MirroredTypeException</code>	Excepción arrojada cuando una aplicación trata de acceder al objeto <code>Class</code> de un TypeMirror.
<code>javax.lang.model.type.MirroredTypesException</code>	Excepción arrojada cuando se trata de acceder a una secuencia de objetos <code>Class</code> de a un TypeMirror (p. ej. un array <code>Class[]</code> elemento de un tipo anotación).

Las existencia de los “Mirror” no tiene una justificación conceptual, como sí la tenía la separación de conceptos entre Elementos y Tipos, si no que es provocada por circunstancias “subterráneas”. Para entender la necesidad de introducir la figura de los “Mirrors” es muy importante tener en cuenta un detalle trascendental: **el procesamiento de anotaciones se lleva a cabo en tiempo de compilación.**

El compilador Java no carga la definición de las clases de forma secuencial o completa, si no siguiendo complejas estrategias de optimización. Esto implica que, durante el proceso de compilación, en el entorno en el que se realiza el procesamiento de anotaciones, **no es posible disponer de cierta información de las clases normalmente almacenada en tiempo de ejecución en una instancia de `Class`**. Por este motivo, se creó el **concepto de los “Mirrors” o “imágenes especulares”**: algo similar a lo que sería un objeto proxy, pero parcial; a efectos prácticos, los “Mirrors” serían más bien como una **fachada segura a la que se le puede pedir información** con la certeza de que no nos encontraremos con algún problema debido a encontrarnos en tiempo de compilación.

Como se introdujeron de mala gana, se crearon sólo dos “interfaces Mirror” únicamente para los objetos que podían dar problemas por no estar cargados por el compilador: la meta-information de las anotaciones (de aquí surgió `AnnotationMirror`) y la información de tipos a través de `Class` (de ahí `TypeMirror`). Debido a que la necesidad de los “Mirrors” se debe a la implementación del compilador Java, la API de procesamiento JSR 269 ha heredado estas “interfaces Mirror” y sus excepciones de la Mirror API, además con los mismos nombres.

Por tanto, **deben tratar de no utilizarse las facilidades de la API de reflexión siempre que la Language Model API ofrezca una alternativa para recuperar la información que necesitamos** para realizar el procesamiento de anotaciones, ya que, en caso de tratar de acceder a una información de reflexión a través del objeto `Class` y dicha información no estar disponible por no haber sido cargada por el compilador, nos encontraremos con que en dicho punto del código nuestro entorno de compilación arrojará una `MirroredTypeException` (o `MirroredTypesException` si se trata de acceder a un array o colección de objetos `Class`).

El ejemplo más claro de los posibles problemas que podemos tener al intentar acceder a información no disponible en tiempo de compilación lo tenemos al tratar de recuperar información sobre las anotaciones que anotan un determinado elemento. Esto puede hacerse, entre otras vías, usando dos métodos de la interfaz **Element**:

Métodos de recuperación de anotaciones de <code>javax.lang.model.element.Element</code>	
Métodos	Descripción
<code><A extends Annotation> A getAnnotation(Class<A> annotationType)</code>	Usa reflexión. CUIDADO .
<code>List<? extends AnnotationMirror> getAnnotationMirrors()</code>	Usa "Mirrors". SEGURO .

NOTA: En la interfaz de utilidad `Elements` está `getAllAnnotationMirrors()` que nos da todas las anotaciones sobre el elemento, incluyendo aquellas indirectamente presentes vía herencia.

Ejemplo: Procesamiento erróneo de un tipo anotación con un elemento `Class`

Supongamos el tipo anotación `@AnotacionElementoClase` con un elemento `Class`:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface AnotacionElementoClase {
    Class elementoClase() default java.lang.Object.class;
}
```

Si ahora escribiéramos un procesador que usara `getAnnotation` (el método que usa reflexión y, por tanto, peligroso en este contexto de tiempo de compilación), podríamos escribir:

```
// tratamos de recuperar el tipo anotación que queremos procesar
AnotacionElementoClase anotacion = tipoElem.getAnnotation(AnotacionElementoClase.class);
if (anotacion != null) {
    // clase anotada con la anotación que queremos procesar -> mostramos su información
    msg.printNotice("Clase anotada: " + tipoElem.getQualifiedName());
    msg.printNotice("Anotación: @" + AnotacionElementoClase.class.getCanonicalName());
    msg.printNotice("Valor: " + anotacion.elementoClase());
}
```

Si ahora anotamos una clase cualquiera con `@AnotacionElementoClase` tal que así:

```
@AnotacionElementoClase(elementoClase=Integer.class)
public class ClaseAnotadaElementoClase { ... }
```

Al ejecutar nuestro procesador, ya que la información de `Integer.class` no ha sido cargada por el compilador, al tratar de acceder a ella obtendríamos la `MirroredTypeException`:

```
An annotation processor threw an uncaught exception.
Consult the following stack trace for details.
javax.lang.model.type.MirroredTypeException: Attempt to access Class object for TypeMirror
java.lang.Integer
... resto de la traza de la pila...
    at com.sun.tools.javac.main.JavaCompiler.processAnnotations(JavaCompiler.java:1108)
    at com.sun.tools.javac.main.JavaCompiler.compile(JavaCompiler.java:824)
    at com.sun.tools.javac.main.Main.compile(Main.java:439)
    at com.sun.tools.javac.main.Main.compile(Main.java:353)
    at com.sun.tools.javac.main.Main.compile(Main.java:342)
    at com.sun.tools.javac.main.Main.compile(Main.java:333)
    at com.sun.tools.javac.Main.compile(Main.java:76)
    at com.sun.tools.javac.Main.main(Main.java:61)
```

En el código anterior se puede ver como hay una línea que curiosamente emplea reflexión (`AnotacionElementoClase.class.getCanonicalName()`) y que funciona sin problemas. Y es que, aunque el compilador no carga la información de todas las clases, sí que carga la mayoría necesaria para lograr que la compilación habitual no falle. Los casos en los que se sabe que esto da problemas son excepcionales y las “interfaces Mirror” son sólo para dichos casos.

Ejemplo: Procesamiento correcto de un tipo anot. con un elemento Class

Si cambiamos a `getAnnotationMirrors()`, sin reflexión y, por tanto, seguro en tiempo de compilación, queda un código más farragoso, ya que ahora no tendremos una instancia del tipo anotación para acceder directamente al valor del elemento (`anotacion.elementoClase()`), si no “Mirrors” con colecciones sobre las que iterar, primero para encontrar la anotación entre una colección de `AnnotationMirror`, y luego para encontrar el elemento buscado:

```
// tratamos de recuperar la declaración del tipo anotación que queremos procesar
List<? extends AnnotationMirror> annotationMirrors = tipoElem.getAnnotationMirrors();

// buscamos la anotación de nuestro tipo @AnotacionElementoClase
for (AnnotationMirror annotationMirror : annotationMirrors) {

    if (!AnotacionElementoClase.class.getCanonicalName()
        .equals(annotationMirror.getAnnotationType().toString())) {
        // anotación que no es del tipo que buscamos -> la ignoramos
        continue;
    }

    // anotación del tipo que buscamos -> recuperamos el valor de elementoClase

    // obtenemos los valores de la anotación en un map
    Map<? extends ExecutableElement, ? extends AnnotationValue> mapValores = null;
    mapValores = annotationMirror.getElementValues();

    // iteramos sobre las claves de la anotación
    for (Map.Entry<? extends ExecutableElement, ? extends AnnotationValue> entrada : mapValores.entrySet()) {

        if !"elementoClase".equals(entrada.getKey().getSimpleName().toString()) {
            // no es la clave del elemento que buscamos -> lo ignoramos
            continue;
        }

        // clave que buscamos -> recuperamos su valor de clase
        // NOTA 1: nótese que el valor del elemento NO viene en un Class, si no en un TypeMirror
        // NOTA 2: el primer getValue() recupera un objeto AnnotationValue y
        // el segundo getValue() el Object que realmente es el valor del elemento buscado
        AnnotationValue valoresElementosClase = entrada.getValue();
        TypeMirror elementoClase = (TypeMirror) valoresElementosClase.getValue();

        // hemos encontrado la anotación y su elementoClase -> mostramos su información
        messenger.printMessage(Kind.NOTE, "Clase anotada: " + tipoElem.getQualifiedName());
        messenger.printMessage(Kind.NOTE, "Anotacion: @" + AnotacionElementoClase.class.getCanonicalName());
        messenger.printMessage(Kind.NOTE, "Valor (clase): " + elementoClase.getClass());
        messenger.printMessage(Kind.NOTE, "Valor: " + elementoClase);
    }
}
```

No obstante, podría conseguirse un código igual de compacto al original si refactorizamos esta búsqueda tan engorrosa a través de un método auxiliar. Y lo mejor de todo es que usando `getAnnotationMirrors()` el procesador no corre peligro de arrojar la `MirroredTypeException`, por lo que obtenemos la salida correcta deseada en un principio:

```
Note: Clase anotada: anotaciones.procesamiento.java67.pruebas.mirrors.ClaseAnotadaElementoClase
Note: Anotacion: @anotaciones.procesamiento.java67.pruebas.mirrors.AnotacionElementoClase
Note: Valor: java.lang.Integer
```

getAnnotation: Un método especial.

El error obtenido usando `getAnnotation`, no obstante, no debería sorprendernos en absoluto, ya que está reflejado en la documentación. En la página del [javadoc oficial](#) de la interfaz `Element`, en el método `getAnnotation` dice lo siguiente (traducido del original):

```
<A extends Annotation> A getAnnotation(Class<A> annotationType)
```

Devuelve la anotación de este elemento para el tipo especificado si dicha anotación está presente, si no `null`. La anotación podría ser heredada o estar directamente presente en este elemento.

La anotación retornada por este método podría contener un elemento cuyo valor de tipo es `Class`. Este valor no puede ser retornado directamente: la información necesaria para localizar y cargar una clase (como por ejemplo el *class loader* a utilizar) no está disponible, y la clase podría no ser localizable en absoluto. Intentar leer un objeto `Class` invocando el método pertinente sobre la anotación retornada resultará en una `MirroredTypeException`, de la cual podría extraerse el correspondiente `TypeMirror`. De forma similar, intentar leer un elemento de tipo `Class[]` resultará en una `MirroredTypesException`.

Así pues, los desarrolladores estaban sobre aviso. Sin embargo, en la documentación de este método hay una nota especial (algo muy poco habitual en la documentación de las APIs oficiales Java) que supone la escueta explicación de la documentación oficial al motivo de incluir las “interfaces Mirror” desde un principio, además de poner de manifiesto que el método `getAnnotation` supone una excepción al usar reflexión:

Nota: Este método es diferente de los demás en esta y otras interfaces relacionadas. Opera sobre información reflexiva en tiempo de ejecución -- representaciones de tipos anotación actualmente cargadas por la MV -- en lugar de sobre las representaciones especulares definidas y usadas a través de estas interfaces. Por consiguiente, llamar a métodos sobre la anotación retornada podrá arrojar muchas de las excepciones arrojadas por un objeto anotación retornado por reflexión. Este método está dirigido a códigos cliente escritos para operar sobre un conjunto fijo y conocido de tipos anotación.

Es decir, que `getAnnotation` sólo debe ser usado por desarrolladores que comprendan lo explicado en este apartado y que, sabiendo que los tipos anotación (el aludido “conjunto fijo y conocido de tipos anotación”) que van a procesar no tienen elementos `Class` o `Class[]` y que no corren peligro de toparse con la `MirroredTypeException`, quieran aprovecharse de la enorme comodidad que ofrece este método a la hora de escribir un código más sencillo.

También se menciona que “intentar leer un objeto `Class` invocando el método pertinente sobre la anotación retornada resultará en una `MirroredTypeException`, de la cual podría extraerse el correspondiente `TypeMirror`”, ¿a qué se refiere con esto? Pues a que si se intenta leer la información de `Integer.class` dicha información sabemos que no existe, pero sí existirá el `TypeMirror` correspondiente con el valor `java.lang.Integer`, que es lo que recuperamos en el ejemplo que usa `getAnnotationMirrors()`. Por eso es que en la `MirroredTypeException` el entorno de compilación ~~nos~~ puede ofrecer el `TypeMirror` que necesitamos!!! Veamos como funciona esto con el ejemplo de una alternativa más para obtener el valor del elemento `Class` de un tipo anotación.

Ejemplo: Procesamiento correcto alternativo de un tipo anotación con un elemento Class

Volviendo al código inicial que daba error, si capturamos la `MirroredTypeException` y accedemos al `TypeMirror` que nos devuelve, podemos obtener el valor que buscábamos, aunque, en lugar de llegarnos como un objeto `Class`, nos llega la información del valor del elemento `Class` como un objeto `TypeMirror`. El nuevo código alternativo quedaría así:

```
if (anotacion != null) {  
  
    // clase anotada con la anotación que queremos procesar -> mostramos su información  
    messenger.printMessage(Kind.NOTE,  
        "Clase anotada: " + tipoElem.getQualifiedName());  
    messenger.printMessage(Kind.NOTE,  
        "Anotacion: @" + AnotacionElementoClase.class.getCanonicalName());  
    try {  
        messenger.printMessage(Kind.NOTE,  
            "Valor: " + anotacion.elementoClase());  
    } catch (MirroredTypeException mtex) {  
        messenger.printMessage(Kind.NOTE,  
            "Valor (recuperado de la MirroredTypeException): " + mtex.getTypeMirror());  
    } catch (Exception ex) {  
        messenger.printMessage(Kind.ERROR, ex.toString());  
    }  
}
```

Se captura la excepción y se utiliza el `TypeMirror` devuelto por `mtex.getTypeMirror()`. Este método alternativo de recuperar el valor `Class` de un tipo anotación también funciona:

```
Note: Clase anotada: anotaciones.procesamiento.java67.pruebas.mirrors.ClaseAnotadaElementoClase  
Note: Anotacion: @anotaciones.procesamiento.java67.pruebas.mirrors.AnotacionElementoClase  
Note: Valor (recuperado de la MirroredTypeException): java.lang.Integer
```

Esta misma estrategia es posible, como se comenta en la documentación oficial, también para `Class[]`, pero capturando la excepción `MirroredTypesException`, y recuperando la colección de `TypeMirror` de cada una de las clases. Realizando un cambio en el diseño de nuestro tipo anotación para que contenga un elemento array `Class[]` y cambiándole el nombre a `@AnotacionElementosClase`, con un código muy similar al anterior, para la anotación `@AnotacionElementosClase(elementosClase={Integer.class, Long.class})` obtenemos:

```
Note: Clase anotada: anotaciones.procesamiento.java67.pruebas.mirrors.ClaseAnotadaElementosClase  
Note: Anotacion: @anotaciones.procesamiento.java67.pruebas.mirrors.AnotacionElementosClase  
Note: Valor (recuperado de la MirroredTypesException): [java.lang.Integer, java.lang.Long]
```

No obstante, **no se considera buena práctica de programación utilizar la captura de excepciones de forma sistemática para implementar lógica de negocio. Así pues, consideramos esta estrategia como alternativa y no recomendable.**

IMPORTANTE: Además de lo anterior, en Java SE 6, acceder a un elemento `Class[]` podría arrojar erróneamente una `MirroredTypeException` sólo para el primer elemento del array, como se explica en este blog: <http://blog.retep.org/2009/02/13/getting-class-values-from-annotations-in-an-annotationprocessor/> Esto es un bug de la implementación corregido en Java SE 7. En Java SE 6, se impone pues aún más `getAnnotationMirrors()` para recuperar la información.

En las siguientes páginas vamos a comentar cómo se lograría esto mismo mediante el uso de `getAnnotationMirrors()` y, sabiendo que requiere mucho trasiego de código, y partiendo del código completo original, ofreceremos al lector un ejemplo del código de un método auxiliar que simplificaría dicho trasiego y facilitaría enormemente el uso de los `AnnotationMirror`.

Si implementamos esto mismo usando `getAnnotationMirrors()`, que es lo preferible, podemos utilizar un código similar al siguiente, que inicialmente sale muy engoroso:

```
// tratamos de recuperar la declaración del tipo anotación que queremos procesar
List<? extends AnnotationMirror> annotationMirrors = tipoElem.getAnnotationMirrors();

// buscamos la anotación de nuestro tipo @AnotacionElementosClase
for (AnnotationMirror annotationMirror : annotationMirrors) {

    if (!AnotacionElementosClase.class.getCanonicalName()
        .equals(annotationMirror.getAnnotationType().toString())) {

        // anotación que no es del tipo que buscamos -> la ignoramos
        continue;
    }

    // anotación del tipo que buscamos -> recuperamos el valor de elementosClase

    // obtenemos los valores de los elementos de la anotación en un Map
    Map<? extends ExecutableElement, ? extends AnnotationValue> mapValores = null;
    mapValores = annotationMirror.getElementValues();

    // iteramos sobre las entradas de los valores de los elementos de la anotación
    for (Map.Entry<? extends ExecutableElement, ? extends AnnotationValue> entrada : mapValores.entrySet()) {

        if !"elementosClase".equals(entrada.getKey().getSimpleName().toString()) {

            // no es la clave del elemento que buscamos -> lo ignoramos
            continue;
        }

        // clave que buscamos -> recuperamos su valor de clase
        // NOTA 1: nótese que el valor del elemento NO viene en un Class normal,
        // si no en una clase interna: com.sun.tools.javac.code.Attribute$Class
        // NOTA 2: el primer getValue() recupera un objeto AnnotationValue y
        // el segundo getValue() los atributos de clase con la información buscada
        AnnotationValue valoresElementosClase = entrada.getValue();
        @SuppressWarnings("unchecked")
        List<AnnotationValue> elementosClase =
            (List<AnnotationValue>) valoresElementosClase.getValue();
        ArrayList<String> nombresCanonicosClases = new ArrayList<>();
        for (AnnotationValue elementoClase : elementosClase) {
            DeclaredType claseDeclaredType = (DeclaredType) elementoClase.getValue();
            nombresCanonicosClases.add(claseDeclaredType.toString());
        }

        // hemos encontrado la anotación y su elementoClase -> mostramos su información
        messenger.printMessage(Kind.NOTE, "Clase anotada: " + tipoElem.getQualifiedName());
        messenger.printMessage(Kind.NOTE,
            "Anotacion: @" + AnotacionElementosClase.class.getCanonicalName());
        messenger.printMessage(Kind.NOTE, "Valor: " + nombresCanonicosClases);
    }
}
}
```

El por qué los elementos de clase se han down-casteado a `List<AnnotationValue>` es porque como valor del tipo elemento `Class[]`, en lugar de un `AnnotationValue` con un `TypeMirror`, lo que recibimos inicialmente es un objeto `AnnotationValue` con una lista `com.sun.tools.javac.util.List<com.sun.tools.javac.code.Attribute$Class>`. El tipo componente de la lista no son `TypeMirror`, si no una clase interna `Attribute$Class` de una oscura clase de implementación llamada `Attribute`.

Tanto dicha clase **Attribute** como su clase interna **Attribute\$Class** implementan la interfaz **AnnotationValue**, por lo que lo que realmente estamos recibiendo como valor del elemento de tipo **Class[]** de nuestro tipo anotación es una **AnnotationValue** que, al recuperar su valor con **getValue()**, devuelve una **List<AnnotationValue>**.

Si obtenemos el valor de cada elemento **Class** llamando al **getValue()** de dichos **AnnotationValue** nos devuelve un **com.sun.tools.javac.code.Type\$ClassType**, que es una clase que implementa la interfaz **DeclaredType**. Por ello es que podemos hacer el otro downcast que aparece hacia el final del código de forma segura.

Y de los tipos de la interfaz **DeclaredType** podemos extraer el nombre canónico de las clases que queríamos y presentarlo por pantalla. Todo este resultado es bastante curioso y difiere de lo que esperábamos en un principio. No obstante, igualmente nos proporciona la información que necesitamos. La salida obtenida por el código anterior es la siguiente:

```
Note: Clase anotada: anotaciones.procesamiento.java67.pruebas.mirrors.ClaseAnotadaElementosClase
Note: Anotacion: @anotaciones.procesamiento.java67.pruebas.mirrors.AnotacionElementosClase
Note: Valor: [java.lang.Integer, java.lang.Long]
```

Como vemos, utilizar el método **getAnnotationMirrors()** conlleva un enorme esfuerzo de codificación debido al enorme trasiego que es necesario para dar con el valor buscado. No obstante, parte del código podría refactorizarse a un método auxiliar que podría invocarse más fácilmente. Supongamos que definimos un método auxiliar **getValorElemento** que, dada una lista de **AnnotationMirror** sea capaz de extraer de ella el valor de un elemento de un tipo anotación dado. Siendo así, el código del ejemplo anterior se quedaría en un fragmento mucho más pequeño y sencillo, similar a lo siguiente:

```
// tratamos de recuperar la declaración del tipo anotación que queremos procesar
List<? extends AnnotationMirror> annotationMirrors = tipoElem.getAnnotationMirrors();

List<AnnotationValue> elementoClases =
    getValorElemento(annotationMirrors, AnotacionElementosClase.class, "elementosClase");

ArrayList<String> nombresCanonicosClases = new ArrayList<>();
for (AnnotationValue elementoClase : elementoClases) {
    DeclaredType claseDeclaredType = (DeclaredType) elementoClase.getValue();
    nombresCanonicosClases.add(claseDeclaredType.toString());
}

// hemos encontrado la anotación y su elementoClase -> mostramos su información
messager.printMessage(Kind.NOTE, "Clase anotada: " + tipoElem.getQualifiedName());
messager.printMessage(Kind.NOTE,
    "Anotacion: @" + AnotacionElementosClase.class.getCanonicalName());
messager.printMessage(Kind.NOTE, "Valor: " + nombresCanonicosClases);
```

Aún hay que iterar sobre **elementoClases**, porque, si imprimimos directamente su valor obtenemos por la salida del compilador **java.lang.Integer.class, java.lang.Long.class**. Es decir: los nombres de las clases, sí, pero terminando en **.class**, que no es lo que queremos. No obstante, esto es un mal muy menor viendo la dramática reducción del código gracias a usar **getValorElemento**. De hecho, si el elemento no fuera un array, si no un **Class**, como en el primer ejemplo, sería aún más directo, ya que sabemos que nos llega un **TypeMirror**:

```
TypeMirror valor = getValorElemento(annotationMirrors, AnotacionElementosClase.class, "elementoClase");
```

El código de `getValorElemento`, que probablemente el lector encontrará interesante utilizar como base para implementar su propio método auxiliar, es el siguiente:

```
/**  
 * Obtiene el valor del elemento <code>nOMBREELEMENTO</code> para un <code>TIPOANOTACION</code>  
 * dentro de una lista de <code>annotationMirrors</code>.  
 * El parámetro de tipo R sirve para indicar el tipo del resultado del valor del elemento a  
 * retornar.  
 * De esta forma, el casting se hace dentro del método y no es necesario hacerlo en la llamada.  
 * @param annotationMirrors lista donde buscar la anotación del <code>TIPOANOTACION</code> buscado.  
 * @param tipoAnotacion tipo de la anotación a buscar en la lista de  
<code>annotationMirrors</code>.  
 * @param nombreElemento nombre del elemento cuyo valor queremos recuperar.  
 * @return valor del elemento de la anotación del tipo buscado, o <code>null</code> si no se  
encuentra.  
 */  
@SuppressWarnings("unchecked")  
private <R> R getValorElemento(  
    List<? extends AnnotationMirror> annotationMirrors,  
    Class<? extends Annotation> tipoAnotacion,  
    String nombreElemento) {  
  
    // variable resultado  
    R valorElemento = null;  
  
    // iteramos por las diferentes anotaciones en busca  
    // de la anotación del tipo que buscamos  
    forExterior:  
    for (AnnotationMirror annotationMirror : annotationMirrors) {  
  
        if (!tipoAnotacion.getCanonicalName()  
            .equals(annotationMirror.getAnnotationType().toString())) {  
  
            // anotación que no es del tipo que buscamos -> la ignoramos  
            continue;  
        }  
  
        // anotación del tipo buscado -> buscamos y recuperamos el valor del elemento  
  
        // obtenemos los valores de los elementos de la anotación en un Map  
        Map<? extends ExecutableElement, ? extends AnnotationValue> mapValores = null;  
        mapValores = annotationMirror.getElementValues();  
  
        // iteramos sobre las entradas de los valores de los elementos de la anotación  
        for (Map.Entry<? extends ExecutableElement, ? extends AnnotationValue>  
            entrada : mapValores.entrySet()) {  
  
            if (!nombreElemento.equals(entrada.getKey().getSimpleName().toString())) {  
  
                // no es la clave del elemento que buscamos -> la ignoramos  
                continue;  
            }  
  
            // clave que buscamos -> recuperamos el AnnotationValue del elemento  
            AnnotationValue elementoAnnotationValue = entrada.getValue();  
  
            // el resultado que buscamos es, a su vez, el valor del AnnotationValue  
            valorElemento = (R) elementoAnnotationValue.getValue();  
  
            // habiendo encontrado el resultado, podemos salir del for exterior  
            break forExterior;  
        }  
    }  
  
    // finalmente, devolvemos el resultado  
    return valorElemento;  
}
```

15.3.3.6.- Uso del patrón Visitor.

Cuando se desarrolla un procesador de anotaciones es común tener que implementar diversas lógicas de proceso según el tipo de elemento procesado. Por lo que se hace habitual tener construcciones condicionales de **if** con una condición por cada tipo de elemento:

Construcciones condicionales de Elementos	Construcciones condicionales de Tipos
<pre>if (elem.getKind() == ElementKind.CLASS) { // ...procesamiento del elemento clase... } else if (elem.getKind() == ElementKindINTERFACE) { // ...procesamiento del elemento interfaz... } else if ...</pre>	<pre>if (tipo.getKind() == TypeKind.DECLARED) { // ...procesamiento del tipo declarado... } else if (tipo.getKind() == TypeKind.ARRAY) { // ...procesamiento del tipo array... } else if ...</pre>

El [patrón de diseño Visitor](#) permite separar la lógica implementada de la estructura de los objetos procesados, teniendo como una de sus principales ventajas el poder añadir a la jerarquía de objetos procesados nuevas operaciones sin modificar los objetos originales. Aunque en la Language Model API las interfaces Visitor tienen métodos especializados **visitX** (como puntos apropiados para su redefinición), sí que incorporan el método genérico **visit**.

Y es que **la verdadera utilidad de las interfaces Visitor es proporcionar una forma cómoda y sencilla de separar la lógica de procesamiento de anotaciones en métodos específicos para determinados tipos de entidades.**

Como ya veremos, esos tipos de entidades pueden ser los tipos de elementos (interfaz **ElementVisitor**), los tipos de los valores de los elementos de un tipo anotación (interfaz **AnnotationValueVisitor**) y los distintos tipos de datos de Java (**TypeVisitor**).

La Language Model API, al modelar las entidades del lenguaje Java, es una de las más afectadas cuando se introducen en el mismo nuevas construcciones, expresiones o tipos. Y de toda la Language Model API, los Visitor son las que más se ven afectados por estos cambios. Al extremo de que, tal y como han sido diseñados, y como hemos visto en la presentación de los Visitor del paquete **javax.lang.element.util**, se ha hecho necesario incluir una versión de sus implementaciones para cada una de las últimas versiones de la plataforma Java, habiendo clases XVisitor6, XVisitor7, XVisitor8, etcétera.

IMPORTANTE: Los Visitor de una versión antigua de Java, cuando encuentren un tipo de construcción que no reconocen por ser de una versión más moderna, llamarán al método **visitUnknown** que, por defecto, arrojará una excepción **UnknownElementException**. Es **responsabilidad de los desarrolladores de los procesadores de anotación redefinir el método visitUnknown si quieren que la política de los Visitor en dicho caso sea, por ejemplo, ignorar los tipos de construcción no reconocidos sin dar ningún fallo, para lo cual bastaría simplemente con redefinir visitUnknown con su cuerpo vacío**. De hecho, esa es la práctica recomendada en la documentación oficial: [heredar de una implementación de Visitor redefiniendo los métodos oportunos](#).

A continuación, ahondaremos en cada uno de los tipos de Visitor disponibles para el procesamiento de anotaciones JSR 269 tratando de poner de manifiesto algunas de sus posibles aplicaciones, así como las principales características y ventajas de cada uno de ellos.

15.3.3.6.1.- ElementVisitor.

Para visitar los elementos del lenguaje Java según su tipo de interfaz disponemos de los **ElementVisitor**, que es, sin duda, el tipo de Visitor más utilizado, ya que las diferentes lógicas de procesamiento suelen diferenciarse normalmente según el tipo de elemento a tratar.

Los **ElementVisitor** vienen en varias modalidades, cada una de las cuales tiene sus particularidades propias, por lo que, según la lógica y estructura concretas de la lógica de procesamiento que se haya elegido implementar, podrá ser más adecuada una modalidad que otra. Las **modalidades de ElementVisitor disponibles** son las siguientes:

ElementVisitor<R,P>	
visit(Element,P):R	
visit(Element):R	
visitPackage(PackageElement,P):R	
visitType(TypeElement,P):R	
visitVariable(VariableElement,P):R	
visitExecutable(ExecutableElement,P):R	
visitTypeParameter(TypeParameterElement,P):R	
visitUnknown(Element,P):R	

Modalidades de ElementVisitor - Características principales	
Modalidad	Características principales
AbstractElementVisitor	Clase base raíz para la implementación de ElementVisitor .
SimpleElementVisitor	Visita por interfaz de elemento (Package, Type, Variable, Executable y TypeParameter) y ejecuta una acción por defecto dada por defaultAction . Necesitaremos comprobar que en la interfaz de elemento nos encontramos con el ElementKind que queremos procesar.
ElementKindVisitor	Visita según el ElementKind y ejecuta una acción por defecto dada por defaultAction . Al ser tan especializado, no será necesario comprobar que nos encontramos con el ElementKind a procesar.
ElementScanner	Visita por interfaz de elemento (Package, Type, Variable, Executable y TypeParameter). Necesitaremos comprobar que en la interfaz de elemento nos encontramos con el ElementKind que queremos procesar. Además, este tipo de Visitor tiene la cualidad de poder visitar no sólo el elemento en cuestión, si no además todos sus subelementos con su método scan , que además puede ser redefinido para efectuar comportamientos específicos.

SimpleElementVisitor es más adecuado cuando la lógica que se desea implementar es la misma para todos los tipos de elemento y se pueda implementar redefiniendo su método protegido **defaultAction**. Por ejemplo, si lo que queremos es simplemente imprimir sus nombres por la salida del compilador. También será adecuado cuando el tipo anotación sea muy sencillo y se procese de la misma forma sea cual sea el tipo de elemento que se anote. Esto es así porque la idea detrás de **SimpleElementVisitor** es que únicamente haya que redefinir su método **defaultAction**, que será el que implementará la lógica de procesamiento, ya que los métodos **visitX** por defecto llamarán a **defaultAction**.

SimpleElementVisitor<R,P>	
SimpleElementVisitor6()	
SimpleElementVisitor6(R)	
defaultAction(Element,P):R	
visitPackage(PackageElement,P):R	
visitType(TypeElement,P):R	
visitVariable(VariableElement,P):R	
visitExecutable(ExecutableElement,P):R	
visitTypeParameter(TypeParameterElement,P):R	

ElementKindVisitor es la modalidad de Visitor más adecuado cuando se desea implementar una lógica especializada al máximo según el tipo de elemento dado por el enumerado **ElementKind**. Su implementación por defecto también llama al método **defaultAction** heredado de **SimpleElementVisitor**. Tiene la ventaja de que, al ser tan especializado gracias a la enorme cantidad de métodos que ofrece, podemos trabajar cómodamente implementando la lógica que sea necesaria en un método en el que sabemos con total exactitud el tipo de elemento con el que estamos tratando. A diferencia del **SimpleElementVisitor**, el **ElementKindVisitor** será normalmente el más apropiado para implementar lógica habitual de la gran mayoría de procesadores de anotación, por lo que en la práctica será el más utilizado, a no ser que la lógica de procesamiento requiera recorrer los subelementos de un elemento; en cuyo caso más específico podría ser interesante considerar el **ElementScanner**.

<<Java Class>>	
G ElementKindVisitor6<R,P>	
●	visitPackage(PackageElement,P):R
●	visitType(TypeElement,P):R
●	visitTypeAsAnnotationType(TypeElement,P):R
●	visitTypeAsClass(TypeElement,P):R
●	visitTypeAsEnum(TypeElement,P):R
●	visitTypeAsInterface(TypeElement,P):R
●	visitVariable(VariableElement,P):R
●	visitVariableAsEnumConstant(VariableElement,P):R
●	visitVariableAsExceptionParameter(VariableElement,P):R
●	visitVariableAsField(VariableElement,P):R
●	visitVariableAsLocalVariable(VariableElement,P):R
●	visitVariableAsParameter(VariableElement,P):R
●	visitVariableAsResourceVariable(VariableElement,P):R
●	visitExecutable(ExecutableElement,P):R
●	visitExecutableAsConstructor(ExecutableElement,P):R
●	visitExecutableAsInstanceInit(ExecutableElement,P):R
●	visitExecutableAsMethod(ExecutableElement,P):R
●	visitExecutableAsStaticInit(ExecutableElement,P):R
●	visitTypeParameter(TypeParameterElement,P):R

El **ElementScanner** es un tipo de Visitor especial que tiene la particularidad de que la implementación por defecto de sus métodos **visitx** llaman recursivamente a su vez a los métodos de visita de los subelementos del elemento actualmente visitado. Esto tiene el resultado práctico de que una llamada de visita sobre un elemento no se limitará a procesar el elemento visitado, si no que tendrá el efecto de disparar el procesamiento de todos los elementos de la jerarquía cuya raíz sea el elemento inicial. Esta propiedad del **ElementScanner** puede resultar de mucho interés cuando estamos procesando, por ejemplo, un tipo anotación cuya lógica procesa clases en bloque, es decir, que requiere que las clases sean procesadas desde el principio hasta el final de forma que no se mezclen invocaciones a métodos **visitx** de elementos de varias clases. Que se procese el elemento clase y todos sus subelementos antes de pasar a la siguiente clase, es un tipo de requisito muy común. En estos casos el **ElementScanner** ofrece la comodidad de ofrecer por defecto el comportamiento de extender las llamadas a los subelementos del elemento inicial.

<<Java Class>>	
G ElementScanner6<R,P>	
F	scan(Iterable<Element>,P):R
●	scan(Element,P):R
F	scan(Element):R
●	visitPackage(PackageElement,P):R
●	visitType(TypeElement,P):R
●	visitVariable(VariableElement,P):R
●	visitExecutable(ExecutableElement,P):R
●	visitTypeParameter(TypeParameterElement,P):R

Aún con todos los tipos de Visitor disponibles, será finalmente el desarrollador el que, atendiendo a las necesidades concretas de los requisitos de su lógica, podría optar por un Visitor que no se ajustara completamente a sus necesidades, pero que tuviera otro tipo de facilidades. También podría completar mediante código las características correspondientes a otro de los Visitor. Por ejemplo, el **ElementKindVisitor**, que es el que hemos dicho que será el más comúnmente utilizado por su granulada especialización de tipos de elementos, podría, si fuera necesario, incorporar la llamada a los subelementos de un elemento, haciendo que no sea necesario pasarse al **ElementScanner**.

Por tanto, en base a lo anterior, la elección particular de un Visitor concreto tampoco es trascendental en modo alguno como para limitar nuestras capacidades a la hora de implementar una cierta lógica. Este hecho lo vamos a poder comprobar en el siguiente ejemplo, en el cual hemos implementado la misma lógica usando todas las modalidades de **ElementVisitor**.

Ejemplo: Procesamiento usando las varias modalidades de ElementVisitor

Supongamos que se desea un tipo anotación para una herramienta de análisis de código que permitiría contabilizar el número total de clases y métodos definidos y calcular su media. Diseñamos el siguiente tipo anotación marcadora para anotar las clases a procesar:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface AnotacionEjemploVisitor { }
```

Computar la media del número de métodos por clase para la herramienta de análisis podría implementarse sin ningún Visitor de una forma similar a la siguiente, donde se van actualizando y mostrando las estadísticas a medida que se encuentran nuevas clases o métodos:

```
// ITERACIÓN SOBRE LOS ELEMENTOS RAÍZ A PROCESAR

// iteramos por los elementos raíz a procesar
for (Element elem : roundEnv.getRootElements()) {

    // COMPROBACIÓN: ELEMENTO ANOTADO CON EL TIPO ANOTACIÓN A PROCESAR

    if (elem.getAnnotation(AnotacionEjemploVisitor.class) == null) {

        // si el elemento no está anotado con el
        // tipo anotación a procesar -> lo ignoramos
        continue;
    }

    // PROCESAMIENTO DEL ELEMENTO ANOTADO

    // *** MÉTODO 1: SIN VISITOR ***

    // COMPROBACIÓN: ELEMENTO DE TIPO CLASE

    // si el elemento no es una clase -> lo ignoramos
    if (elem.getKind() == ElementKind.CLASS) {

        // PROCESAMIENTO DEL ELEMENTO TIPO CLASE

        // habiendo exigido ya que el elemento sea ElementKind.CLASS
        // no hace falta comprobar nada más, ya que ElementKind.CLASS
        // refleja clases y sólo clases (si fuera un tipo enumerado
        // el tipo del elemento sería específicamente ElementKind.ENUM)

        // down-cast
        TypeElement tipoElem = (TypeElement) elem;

        // actualizamos y mostramos las estadísticas sumando la clase
        sumarClaseYMostrarEstadisticas(tipoElem);

        // para cada uno de los de métodos de la clase...
        for (ExecutableElement metodoElem :
            ElementFilter.methodsIn(tipoElem.getEnclosedElements())) {

            // actualizamos y mostramos las estadísticas sumando el método
            sumarMetodoYMostrarEstadisticas(metodoElem);
        }
    }
}
```

La salida es la información esperada, según se actualiza con cada nueva clase o método:

```
Note: Clase encontrada: ClaseAnotadaAnotacionEjemploVisitor1 >>> Clases = 1 / Métodos: 0 / Media: 0.0
Note: Método encontrado: m1() >>> Clases = 1 / Métodos: 1 / Media: 1.0
Note: Método encontrado: m2() >>> Clases = 1 / Métodos: 2 / Media: 2.0
Note: Método encontrado: m3() >>> Clases = 1 / Métodos: 3 / Media: 3.0
Note: Clase encontrada: ClaseAnotadaAnotacionEjemploVisitor2 >>> Clases = 2 / Métodos: 3 / Media: 1.0
Note: Método encontrado: m1() >>> Clases = 2 / Métodos: 4 / Media: 2.0
Note: Método encontrado: m2() >>> Clases = 2 / Métodos: 5 / Media: 2.5
Note: Método encontrado: m3() >>> Clases = 2 / Métodos: 6 / Media: 3.0
Note: Método encontrado: m4() >>> Clases = 2 / Métodos: 7 / Media: 3.5
Note: Método encontrado: m5() >>> Clases = 2 / Métodos: 8 / Media: 4.0
```

Como en el caso de este ejemplo lo que queremos es procesar un tipo anotación sobre elementos de clase (aunque luego se llame también a los subelementos de métodos), podríamos implementar un Visitor sencillo con **SimpleElementVisitor** para realizar tal acción.

Al no adaptarse los requisitos a la característica del **SimpleElementVisitor** de implementar una acción por defecto redefiniendo su método protegido **defaultAction**, nos hemos limitado a seguir el procedimiento habitual con los Visitor de redefinir únicamente los métodos **visitX** que nos interesaban. En este caso, hemos redefinido el método **visitType**.

Suponiendo que, en el código original, el **for** y el **if** de la comprobación de elemento anotado no se modifican, el cuerpo del bloque **PROCESAMIENTO DEL ELEMENTO ANOTADO**, utilizando el nuevo **SimpleElementVisitor** quedaría de la siguiente manera:

```
// *** MÉTODO 2: CON VISITOR SimpleElementVisitor7 ***
// definimos el visitor que queremos utilizar
SimpleElementVisitor7<Void, Void> simpleElementVisitor7 = null;
simpleElementVisitor7 = new SimpleElementVisitor7<Void, Void>() {

    @Override
    public Void visitType(TypeElement tipoElem, Void param) {

        // COMPROBACIÓN: ELEMENTO DE TIPO CLASE

        // si el elemento no es una clase -> lo ignoramos
        if (tipoElem.getKind() == ElementKind.CLASS) {

            // PROCESAMIENTO DEL ELEMENTO TIPO CLASE

            // habiendo exigido ya que el elemento sea ElementKind.CLASS
            // no hace falta comprobar nada más, ya que ElementKind.CLASS
            // refleja clases y sólo clases (si fuera un tipo enumerado
            // el tipo del elemento sería específicamente ElementKind.ENUM)

            // actualizamos y mostramos las estadísticas sumando la clase
            sumarClaseYMostrarEstadisticas(tipoElem);

            // para cada uno de los de métodos de la clase...
            for (ExecutableElement metodoElem :
                ElementFilter.methodsIn(tipoElem.getEnclosedElements())) {

                // actualizamos y mostramos las estadísticas sumando el método
                sumarMetodoYMostrarEstadisticas(metodoElem);
            }
        }
    }
}
```

```

        // devolvemos como resultado el parámetro
        // sólo para conformar la signatura del método
        return param;
    }

};

// pasamos el visitor al elemento que queremos visitar
elem.accept(simpleElementVisitor7, null);

```

Nótese que lo que hemos hecho ha sido definir una instancia de una clase anónima que hereda de `SimpleElementVisitor` redefiniendo `visitType`. Ahora que la lógica que realiza el procesamiento reside en `visitType`, según el contrato de los Visitor y el propio patrón de diseño Visitor, dicho método sólo será invocado cuando el Visitor se pase a elementos que implementen la interfaz `TypeElement`. Para pasar el Visitor instanciado al elemento se utiliza el método `accept`, como se ve en `elem.accept(simpleElementvisitor7, null);`

La salida obtenida con el `SimpleElementVisitor` ofrece idénticos resultados a los del código original. Aunque, a nivel de código parece que sólo nos hemos ahorrado el down-cast y poco más, los beneficios estructurales y de separación del código fuente serán mayores cuantos más tipos de elementos necesitemos procesar de forma separada.

Y el código anterior reducirse aún más sin la necesidad de hacer la comprobación del tipo de elemento (`tipoElem.getKind() == ElementKind.CLASS`) si utilizamos el `ElementKindVisitor`:

```

// *** MÉTODO 3: CON VISITOR ElementKindVisitor7 ***

// definimos el visitor que queremos utilizar
ElementKindVisitor7<Void, Void> elementKindVisitor7 = null;
elementKindVisitor7 = new ElementKindVisitor7<Void, Void>() {

    @Override
    public Void visitTypeAsClass(TypeElement tipoElem, Void param) {

        // PROCESAMIENTO DEL ELEMENTO TIPO CLASE DIRECTAMENTE

        // actualizamos y mostramos las estadísticas sumando la clase
        sumarClaseYMostrarEstadisticas(tipoElem);

        // para cada uno de los de métodos de la clase...
        for (ExecutableElement metodoElem :
                ElementFilter.methodsIn(tipoElem.getEnclosedElements())) {

            // actualizamos y mostramos las estadísticas sumando el método
            sumarMetodoYMostrarEstadisticas(metodoElem);
        }

        // devolvemos como resultado el parámetro
        // sólo para conformar la signatura del método
        return param;
    }

};

// pasamos el visitor al elemento que queremos visitar
elem.accept(elementKindVisitor7, null);

```

Como se ve en el código anterior, la implementación con el **ElementKindVisitor**, ya no hay que comprobar si el **TypeElement** es específicamente una clase, ya que precisamente para eso la clase **ElementKindVisitor** define un método para cada tipo elemento dado por el enumerado **ElementKind**. Así, los métodos genéricos de la interfaz **ElementVisitor** en **ElementKindVisitor** se especializan lo suficiente para poder prescindir de la comprobación `tipoElem.getKind() == ElementKind.CLASS`, puesto que el método **visitTypeAsClass** sólo se llama para instancias de **TypeElement** que sean clases. El método **defaultAction** tampoco se ha redefinido para este Visitor, ya que en lugar de una acción por defecto, lo que queríamos era redefinir la lógica para procesar los elementos de clase.

Así pues, en base a lo anterior, gracias a estar en el método **visitTypeAsClass**, no se requiere el `if (tipoElem.getKind() == ElementKind.CLASS)`. No obstante, aún es necesaria la presencia del bucle **for** para procesar cada uno de sus métodos. Nuestro ejemplo de forma natural se ajusta al tipo de lógica de procesamiento en bloque por clases, ya que se computan las estadísticas de los subelementos de cada una de las clases anotadas. Siendo así, el ejemplo se podría implementar también haciendo uso de un **ElementScanner** de la siguiente manera:

```
// *** MÉTODO 4: CON VISITOR ElementScanner7 ***

// definimos el visitor que queremos utilizar
ElementScanner7<Void, Void> elementScanner7 = null;
elementScanner7 = new ElementScanner7<Void, Void>() {

    @Override
    public Void visitType(TypeElement tipoElem, Void param) {

        // COMPROBACIÓN: ELEMENTO DE TIPO CLASE

        // si el elemento no es una clase -> lo ignoramos
        if (tipoElem.getKind() == ElementKind.CLASS) {

            // PROCESAMIENTO DEL ELEMENTO TIPO CLASE

            // habiendo exigido ya que el elemento sea ElementKind.CLASS
            // no hace falta comprobar nada más, ya que ElementKind.CLASS
            // refleja clases y sólo clases (si fuera un tipo enumerado
            // el tipo del elemento sería específicamente ElementKind.ENUM)

            // actualizamos y mostramos las estadísticas sumando la clase
            sumarClaseYMostrarEstadisticas(tipoElem);
        }

        // antes de devolver el resultado,
        // debemos llamar al método de la superclase
        // para que se siga llevando a cabo la lógica
        // de llamar a los subelementos de este elemento
        // (al hacerlo después de haber procesado el
        // elemento, estaremos procesando el árbol
        // de jerarquía de los elementos en post-order)
        super.visitType(tipoElem, param);

        // devolvemos como resultado el parámetro
        // sólo para conformar la signatura del método
        return param;
    }
}
```

```

@Override
public Void visitExecutable(ExecutableElement ejecutableElem, Void param) {

    // COMPROBACIÓN: ELEMENTO DE TIPO MÉTODO

    // si el elemento no es un método -> lo ignoramos
    if (ejecutableElem.getKind() == ElementKind.METHOD) {

        // PROCESAMIENTO DEL ELEMENTO TIPO MÉTODO

        // actualizamos y mostramos las estadísticas sumando el método
        sumarMetodoYMostrarEstadisticas(ejecutableElem);

    }

    // antes de devolver el resultado,
    // debemos llamar al método de la superclase
    // para que se siga llevando a cabo la lógica
    // de llamar a los subelementos de este elemento
    // (al hacerlo después de haber procesado el
    // elemento, estaremos procesando el árbol
    // de jerarquía de los elementos en post-order)
    super.visitExecutable(ejecutableElem, param);

    // devolvemos como resultado el parámetro
    // sólo para conformar la firma del método
    return param;
}

};

// realizamos la llamada al método scan pasando el elemento a escanear
elementScanner7.scan(elem, null);

```

Este código vuelve a arrojar idénticos resultados a los de todas las implementaciones anteriores. Quizás el **ElementScanner** salga un poco aparatoso en este caso, ya que **hay que volver a hacer la comprobación de tipo de elemento** y **hay que redefinir 2 métodos**. Nótese además que **ambos métodos redefinidos llaman a la implementación original de las superclase con super**. Si no incluimos esta llamada, no se realizarán las llamadas recursivas a los subelementos, ya que no aparecen en el nuevo código de la redefinición. En función de el **super** se haga antes o después del procesamiento, estaremos implementando un recorrido por el árbol de la jerarquía de elementos en pre-order o post-order.

En nuestro caso, implementamos un post-order, ya que, si no, se daría el contrasentido de empezar a contar los métodos antes que las propias clases, y para los métodos de la primera clase, al computar las estadísticas, ocurriría que en la media de métodos por clase habría que dividir el nº métodos contabilizados por 0 clases, dando una media de infinito. Sólo a modo ilustrativo pues, si modificáramos **visitType** para hacer **super.visitType(tipoElem, param)** como primera línea de código, obtendríamos los siguientes resultados:

```

Note: Método encontrado: m1() >>> Clases = 0 / Métodos: 1 / Media: Infinity
Note: Método encontrado: m2() >>> Clases = 0 / Métodos: 2 / Media: Infinity
Note: Método encontrado: m3() >>> Clases = 0 / Métodos: 3 / Media: Infinity
Note: Clase encontrada: ClaseAnotadaAnotacionEjemploVisitor1 >>> Clases = 1 / Métodos: 3 / Media: 3.0
Note: Método encontrado: m1() >>> Clases = 1 / Métodos: 4 / Media: 4.0
Note: Método encontrado: m2() >>> Clases = 1 / Métodos: 5 / Media: 5.0
Note: Método encontrado: m3() >>> Clases = 1 / Métodos: 6 / Media: 6.0
Note: Método encontrado: m4() >>> Clases = 1 / Métodos: 7 / Media: 7.0
Note: Método encontrado: m5() >>> Clases = 1 / Métodos: 8 / Media: 8.0
Note: Clase encontrada: ClaseAnotadaAnotacionEjemploVisitor2 >>> Clases = 2 / Métodos: 8 / Media: 4.0

```

15.3.3.6.2.- AnnotationValueVisitor.

Para poder visitar los valores de los elementos de las anotaciones del lenguaje Java según su tipo de datos disponemos de la interfaz **AnnotationValueVisitor**, que nos permite definir lógicas comunes correspondiéndose según el tipo de datos concreto de los elementos de un tipo anotación.

AnnotationValueVisitor va a ser un tipo de Visitor realmente muy específico y en principio poco utilizado, ya que no es muy común montar la lógica de procesamiento de los valores de un tipo anotación atendiendo exclusivamente a su tipo de datos. No obstante, según sean los requisitos, podría haber algunos casos en los que podría ser de utilidad definir un Visitor atendiendo al criterio del tipo de datos de los elementos de un tipo anotación.

<<Java Interface>>	
I AnnotationValueVisitor<R,P>	
● visit(AnnotationValue,P):R	
● visit(AnnotationValue):R	
● visitBoolean(boolean,P):R	
● visitByte(byte,P):R	
● visitChar(char,P):R	
● visitDouble(double,P):R	
● visitFloat(float,P):R	
● visitInt(int,P):R	
● visitLong(long,P):R	
● visitShort(short,P):R	
● visitString(String,P):R	
● visitType(TypeMirror,P):R	
● visitEnumConstant(VariableElement,P):R	
● visitAnnotation(AnnotationMirror,P):R	
● visitArray(List<AnnotationValue>,P):R	
● visitUnknown(AnnotationValue,P):R	

Las **modalidades de AnnotationValueVisitor** disponibles son las siguientes:

Modalidades de ElementVisitor - Características principales	
Modalidad	Características principales
AbstractAnnotationValueVisitor	Clase base para la implementación de AnnotationValueVisitor .
SimpleAnnotationValueVisitor	Visita por tipo de datos de los valores AnnotationValue de una anotación representada por un AnnotationMirror y ejecuta una acción por defecto dada por su método protegido defaultAction .

SimpleAnnotationValueVisitor es la más básica implementación de **AnnotationValueVisitor** que se puede utilizar, ya que **AbstractAnnotationValueVisitor** es la clase abstracta raíz para facilitar la creación de nuestros propios **AnnotationValueVisitor**.

SimpleAnnotationValueVisitor está pensado para cuando la lógica que se desea implementar es la misma para todos los tipos de datos y se pueda implementar redefiniendo su método protegido **defaultAction**. Por ejemplo, si lo que queremos es simplemente imprimir sus nombres por la salida del compilador. La idea detrás de este Vistor es que sólo haya que redefinir su método **defaultAction**, que será el que implementará la lógica de procesamiento, ya que todos los métodos **visitX** por defecto llamarán a **defaultAction**.

<<Java Class>>	
C SimpleAnnotationValueVisitor6<R,P>	
● SimpleAnnotationValueVisitor6()	
● SimpleAnnotationValueVisitor6(R)	
● defaultAction(Object,P):R	
● visitBoolean(boolean,P):R	
● visitByte(byte,P):R	
● visitChar(char,P):R	
● visitDouble(double,P):R	
● visitFloat(float,P):R	
● visitInt(int,P):R	
● visitLong(long,P):R	
● visitShort(short,P):R	
● visitString(String,P):R	
● visitType(TypeMirror,P):R	
● visitEnumConstant(VariableElement,P):R	
● visitAnnotation(AnnotationMirror,P):R	
● visitArray(List<AnnotationValue>,P):R	

Ejemplo: Procesamiento usando AnnotationValueVisitor

Para explicar el funcionamiento de `AnnotationValueVisitor`, vamos a suponer un tipo anotación sencillo, que sólo tuviera un elemento de tipo numérico entero tal que así:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public interface AnotacionEjemploVisitorValorNumerico {

    int value() default 0;
}
```

Si quisiéramos simplemente sumar el valor total de todos los valores de cada uno de los elementos anotados, podríamos ubicar dicho código en un `AnnotationValueVisitor` redefiniendo el método `visitInt` con un código similar al siguiente:

```
// iteramos por los elementos raíz a procesar
for (final Element elem : roundEnv.getRootElements()) {

    // COMPROBACIÓN: ELEMENTO ANOTADO CON EL TIPO ANOTACIÓN A PROCESAR

    if (elem.getAnnotation(AnotacionEjemploVisitorValorNumerico.class) == null) {

        // si el elemento no está anotado con el
        // tipo anotación a procesar -> lo ignoramos
        continue;
    }

    // PROCESAMIENTO DEL ELEMENTO ANOTADO

    // CON VISITOR SimpleAnnotationValueVisitor7

    // definimos el visitor que queremos utilizar
    AnnotationValueVisitor<Void, Void> annotationValueVisitor7 = null;
    annotationValueVisitor7 = new SimpleAnnotationValueVisitor7<Void, Void>() {

        @Override
        public Void visitInt(int valorInt, Void param) {

            // PROCESAMIENTO DEL VALOR INT

            // acumulamos sobre el resultado
            total = total + valorInt;

            // damos parte del nuevo valor
            messenger.printMessage(Kind.NOTE,
                "visitado: " + elem.getSimpleName() + ": " +
                "valor: " + valorInt + " >> total = " + total);

            // devolvemos como resultado el parámetro
            // sólo para conformar la firma del método
            return param;
        }
    };
}
```

```

// obtenemos una referencia al o los AnnotationMirror
// del tipo anotación que estamos procesando
List<? extends AnnotationMirror> mirrors = elements.getAllAnnotationMirrors(elem);

// obtenemos una referencia al o los AnnotationMirror
// del tipo anotación que estamos procesando
// (dejamos abierta la posibilidad de anotaciones repetidas,
// por lo que este código serviría para Java SE 8+)
for (AnnotationMirror mirror: mirrors) {

    // ¿es del tipo anotación a procesar?
    if (mirror.getAnnotationType().toString().
        equals(AnotacionEjemploVisitorValorNumerico.class.getCanonicalName()))
    {

        // la AnnotationMirror es del tipo anotación
        // que estamos procesando -> pasamos el visitor
        // a todos sus AnnotationValue's

        // obtenemos los valores de la AnnotationMirror
        Map<? extends ExecutableElement, ? extends AnnotationValue> mapValores =
            this.elements.getElementValuesWithDefaults(mirror);

        // pasamos el visitor a toda la colección de valores del mirror
        for (AnnotationValue valor : mapValores.values()) {

            // pasamos el visitor al AnnotationValue
            valor.accept(annotationValueVisitor7, null);
        }
    }
}

} // for Element

```

En el código anterior podemos distinguir varios bloques: uno inicial donde, como suele ser habitual, se ignoran los elementos que no están anotados con el tipo anotación que queremos procesar, otro a continuación donde se define una instancia de una clase anónima que hereda de **SimpleAnnotationValueVisitor** redefiniendo el método **visitInt**, y el bloque final donde se recuperan los **AnnotationMirror** que contienen los **AnnotationValue** que queremos procesar para pasarles el Visitor que hemos creado.

Lo más incómodo del código anterior es que, como **AnnotationValueVisitor** trabaja con **AnnotationMirror**, el código empieza a ponerse farragoso una vez que tenemos que empezar a montar la típica estructura **for-if-Map-for** que suele ser necesario montar con los **AnnotationMirror** para poder alcanzar finalmente los valores a los que queremos llegar.

El código fuente anterior, ante la existencia de 2 clases anotadas con el tipo anotación `@AnotacionEjemploVisitorValorNumerico` con valores 111 y 222, muestra la salida esperada:

```

Note: visitado: ClaseAnotadaAnotacionEjemploVisitorValorNumerico1: valor: 111 >>> total = 111
Note: visitado: ClaseAnotadaAnotacionEjemploVisitorValorNumerico2: valor: 222 >>> total = 333

```

15.3.3.6.3.- TypeVisitor.

Para poder visitar los tipos de los elementos del lenguaje Java disponemos de la interfaz **TypeVisitor**, que nos permite definir lógicas según el tipo concreto de los elementos visitados.

TypeVisitor no va a ser un tipo de Visitor utilizado muy a menudo, ya que no es habitual crear la lógica de procesamiento de atendiendo exclusivamente al tipo representado por su elemento. No obstante, según sean los requisitos que se nos exijan, podría haber casos en los que será de utilidad definir un Visitor atendiendo al tipo de los elementos.

Las **modalidades de TypeVisitor disponibles** son:

<<Java Interface>>	
TypeVisitor<R,P>	
● visit(TypeMirror,P):R	
● visit(TypeMirror):R	
● visitPrimitive(PrimitiveType,P):R	
● visitNull(NullType,P):R	
● visitArray(ArrayType,P):R	
● visitDeclared(DeclaredType,P):R	
● visitError(ErrorType,P):R	
● visitTypeVariable(TypeVariable,P):R	
● visitWildcard(WildcardType,P):R	
● visitExecutable(ExecutableType,P):R	
● visitNoType(NoType,P):R	
● visitUnknown(TypeMirror,P):R	
● visitUnion(UnionType,P):R	

Modalidades de ElementVisitor - Características principales	
Modalidad	Características principales
AbstractTypeVisitor	Clase base abstracta raíz para facilitar la implementación de nuevos TypeVisitor .
SimpleTypeVisitor	Visita según el tipo de un elemento anotado y ejecuta una acción por defecto dada por su método protegido defaultAction .
TypeKindVisitor	Visita según el TypeKind del tipo de un elemento anotado y ejecuta una acción por defecto dada por defaultAction . Al ser tan especializado, no será necesario comprobar que nos encontramos con el TypeKind a procesar.

SimpleTypeVisitor es la implementación sencilla de **TypeVisitor** y la más adecuada cuando la lógica que se desea implementar es la misma para todos los tipos de elemento y lo mejor es implementar una acción por defecto redefiniendo su método protegido **defaultAction**. También será adecuado cuando el tipo anotación sea muy sencillo y se procese de la misma forma sea cual sea el tipo que represente el elemento anotado. Esto es así porque la idea detrás de **SimpleTypeVisitor** es que únicamente haya que redefinir su método **defaultAction**, que será el que implementará la lógica de procesamiento o acción por defecto, ya que los métodos **visitX** por defecto llamarán a **defaultAction**.

<<Java Class>>	
SimpleTypeVisitor6<R,P>	
◊ SimpleTypeVisitor6()	
◊ SimpleTypeVisitor6(R)	
◊ defaultAction(TypeMirror,P):R	
● visitPrimitive(PrimitiveType,P):R	
● visitNull(NullType,P):R	
● visitArray(ArrayType,P):R	
● visitDeclared(DeclaredType,P):R	
● visitError(ErrorType,P):R	
● visitTypeVariable(TypeVariable,P):R	
● visitWildcard(WildcardType,P):R	
● visitExecutable(ExecutableType,P):R	
● visitNoType(NoType,P):R	

TypeKindVisitor es la modalidad de Visitor más adecuado cuando se desea implementar una lógica especializada al máximo según la clase del tipo dada por **TypeKind**. Su implementación por defecto también llama al método protegido **defaultAction** heredado de **SimpleTypeVisitor**. Tiene la ventaja de que, al ser tan especializado gracias a su enorme cantidad de métodos, que podemos trabajar cómodamente implementando la lógica que sea necesaria en un método en el que sabemos con total exactitud el tipo con el que estamos tratando. El **TypeKindVisitor** será normalmente el elegido a la hora de implementar Visitors por tipo, ya que, al encontrarnos en un método con un tipo especializado, nos resultará muy cómodo poder trabajar directamente con el tipo sin tener que comprobar su tipo real comparándolo con un valor de **TypeKind**.

<<Java Class>>	
TypeKindVisitor6<R,P>	
● visitPrimitive(PrimitiveType,P):R	
● visitPrimitiveAsBoolean(PrimitiveType,P):R	
● visitPrimitiveAsByte(PrimitiveType,P):R	
● visitPrimitiveAsShort(PrimitiveType,P):R	
● visitPrimitiveAsInt(PrimitiveType,P):R	
● visitPrimitiveAsLong(PrimitiveType,P):R	
● visitPrimitiveAsChar(PrimitiveType,P):R	
● visitPrimitiveAsFloat(PrimitiveType,P):R	
● visitPrimitiveAsDouble(PrimitiveType,P):R	
● visitNoType(NoType,P):R	
● visitNoTypeAsVoid(NoType,P):R	
● visitNoTypeAsPackage(NoType,P):R	
● visitNoTypeAsNone(NoType,P):R	

Ejemplo: Procesamiento usando TypeVisitor

Supongamos que se desea un tipo anotación exclusivamente sobre clases para una herramienta de análisis de código que haga que se contabilice el número de campos de tipos primitivos o **String** que se han definido en la clase. Los resultados deberán ofrecerse totalizados y por tipo. Diseñamos el siguiente tipo anotación para anotar las clases a procesar:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface AnotacionEjemploTypeVisitor { }
```

Computar las estadísticas solicitadas en los requisitos con un **SimpleTypeVisitor** podría implementarse de la siguiente forma:

```
// iteramos por los elementos raíz a procesar
for (Element elem : roundEnv.getRootElements()) {

    // COMPROBACIÓN: ELEMENTO CLASE ANOTADO CON EL TIPO ANOTACIÓN A PROCESAR

    if ((elem.getKind() != ElementKind.CLASS)
        || (elem.getAnnotation(AnotacionEjemploTypeVisitor.class) == null)) {

        // si el elemento no es una clase o no está anotado
        // con el tipo anotación a procesar -> lo ignoramos
        continue;
    }

    // down-cast a elemento de tipo (sabemos además que es una clase)
    TypeElement claseElem = (TypeElement) elem;

    // PROCESAMIENTO DEL ELEMENTO ANOTADO

    // IMPLEMENTACIÓN VISITOR SimpleTypeVisitor7

    // definimos el visitor que queremos utilizar
    TypeVisitor<Void, Void> simpleTypeVisitor7 = null;
    simpleTypeVisitor7 = new simpleTypeVisitor7<Void, Void>() {

        @Override
        public Void visitPrimitive(PrimitiveType tipoPrimitivoType, Void param) {

            // CONTEO DE CAMPO PRIMITIVO

            // sumamos uno al contador de campos de tipo primitivo
            totalCamposPrimitivosYStrings++;

            // CONTEO ADICIONAL POR TIPO PRIMITIVO

            // referencia a la clase/kind de tipo
            TypeKind tipo = tipoPrimitivoType.getKind();

            // incrementamos el total según el tipo primitivo
            if (tipo == TypeKind.BOOLEAN) { totalCamposBoolean++; }
            else if (tipo == TypeKind.BYTE) { totalCamposByte++; }
            else if (tipo == TypeKind.CHAR) { totalCamposChar++; }
            else if (tipo == TypeKind.SHORT) { totalCamposShort++; }
            else if (tipo == TypeKind.INT) { totalCamposInt++; }
            else if (tipo == TypeKind.LONG) { totalCamposLong++; }
            else if (tipo == TypeKind.FLOAT) { totalCamposFloat++; }
            else if (tipo == TypeKind.DOUBLE) { totalCamposDouble++; }

            // devolvemos como resultado el parámetro
            // sólo para conformar la firma del método
            return param;
        }

        @Override
```

```

public Void visitDeclared(DeclaredType tipoDeclaredType, Void param) {
    // CONTEO DE CAMPO STRING (SI PROCEDE)

    // incrementamos el total si el campo es de tipo String
    if (tipoDeclaredType.toString().equals("java.lang.String")) {

        // tipo String -> actualizamos los totales
        totalCamposPrimitivosYStrings++;
        totalCamposString++;
    }

    // devolvemos como resultado el parámetro
    // sólo para conformar la firma del método
    return param;
}

};

// CÁLCULO DE ESTADÍSTICAS

// pasamos el visitor a los tipos de los campos del
// elemento clase anotado que tenemos que procesar

// campos de la clase
List<VariableElement> campos = ElementFilter.fieldsIn(claseElem.getEnclosedElements());

// pasamos el visitor a los tipos de todos los campos
for (VariableElement campo : campos) {

    // tipo del campo
    TypeMirror tipoCampo = campo.asType();

    // pasamos el visitor al tipo del campo
    tipoCampo.accept(simpleTypeVisitor7, null);
}

// cuando terminamos, presentamos los resultados
this.messager.printMessage(Kind.NOTE,
    "estadisticas clase " + claseElem.getSimpleName() + ":\n" +
    "Campos de tipo primitivo y String: " + this.totalCamposPrimitivosYStrings + " (TOTAL)\n" +
    "Campos de tipo primitivo boolean: " + this.totalCamposBoolean + "\n" +
    "Campos de tipo primitivo byte: " + this.totalCamposByte + "\n" +
    "Campos de tipo primitivo char: " + this.totalCamposChar + "\n" +
    "Campos de tipo primitivo short: " + this.totalCamposShort + "\n" +
    "Campos de tipo primitivo int: " + this.totalCamposInt + "\n" +
    "Campos de tipo primitivo long: " + this.totalCamposLong + "\n" +
    "Campos de tipo primitivo float: " + this.totalCamposFloat + "\n" +
    "Campos de tipo primitivo double: " + this.totalCamposDouble + "\n" +
    "Campos de tipo primitivo String: " + this.totalCamposString);

// después de mostrar los resultados, hay que resetear
// todos los contadores a 0 para procesar la siguiente clase
this.totalCamposPrimitivosYStrings = 0;
this.totalCamposBoolean = 0;
this.totalCamposByte = 0;
this.totalCamposChar = 0;
this.totalCamposShort = 0;
this.totalCamposInt = 0;
this.totalCamposLong = 0;
this.totalCamposFloat = 0;
this.totalCamposDouble = 0;
this.totalCamposString = 0;

} // for Element

```

El código anterior implementa los requisitos solicitados definiendo una instancia de un Visitor a partir de una clase anónima que hereda de **SimpleTypeVisitor** y que redefine 2 métodos: **visitPrimitive** y **visitDeclared**. Esto es así porque, además de contar los campos de tipo primitivo, hemos de contar los del tipo **String**, que es un tipo declarado con nombre canónico “**java.lang.String**”.

Una vez definido el **SimpleTypeVisitor**, recuperamos la lista de campos de la clase anotada que estamos procesando y aplicamos el Visitor a los tipos de los campos de dicha clase. Finalmente, se muestran los resultados y se resetean los contadores por si hay que procesar alguna clase más, que no se acumulen, dando lugar a estadísticas incorrectas.

Ejecutando el procesamiento, el procesador imprime la siguiente información por la salida del compilador, la cual es correcta según las clases anotadas que se han definido:

```
Note: estadisticas clase ClaseAnotadaAnotacionEjemploTypeVisitor1:  
Campos de tipo primitivo y String: 20 (TOTAL)  
Campos de tipo primitivo boolean: 2  
Campos de tipo primitivo byte: 2  
Campos de tipo primitivo char: 4  
Campos de tipo primitivo short: 2  
Campos de tipo primitivo int: 3  
Campos de tipo primitivo long: 2  
Campos de tipo primitivo float: 2  
Campos de tipo primitivo double: 2  
Campos de tipo primitivo String: 1  
Note: estadisticas clase ClaseAnotadaAnotacionEjemploTypeVisitor2:  
Campos de tipo primitivo y String: 10 (TOTAL)  
Campos de tipo primitivo boolean: 0  
Campos de tipo primitivo byte: 0  
Campos de tipo primitivo char: 3  
Campos de tipo primitivo short: 0  
Campos de tipo primitivo int: 3  
Campos de tipo primitivo long: 0  
Campos de tipo primitivo float: 1  
Campos de tipo primitivo double: 1  
Campos de tipo primitivo String: 2
```

Esta implementación usando **SimpleTypeVisitor** se podría mejorar usando un **TypeKindVisitor** y una clase interna que sirviera para guardar los resultados estadísticos, pudiendo pasarse como parámetro del Visitor.

Hasta ahora no habíamos empleado la **posibilidad que permiten todos los Visitor de configurar sus variables de tipo, ya sea tanto de tipo parámetro como de tipo resultado**, y como se ajusta muy bien a lo que necesitamos para este ejemplo, lo utilizaremos aquí como **ejemplo del uso de los parámetros de tipo <R, P> de los Visitor**.

En código de ejemplo de la página siguiente pues, implementaremos estos mismos requisitos usando un **TypeKindVisitor**, redefiniendo un método **visitPrimitiveAsX** por cada tipo primitivo. Además, definiremos una clase interna **Estadísticas** que nos servirá para almacenar los contadores de resultados estadísticos, lo cual nos evitará tener que resetar los contadores cada vez que vayamos a visitar los tipos de una nueva clase, ya que al Visitor ahora podremos pasarle una nueva instancia de **Estadísticas** para almacenar los resultados de cada nueva clase anotada que procesemos.

El código que implementa los requisitos del ejemplo de conteo de campos de tipos primitivos y **String** en las clases anotadas usando un **TypeKindVisitor** y una clase interna **Estadisticas** para guardar los resultados, así como parámetro de tipo <R, P>, a la vez como tipo parámetro y como tipo resultado del Visitor, quedaría de la siguiente manera:

```
// iteramos por los elementos raíz a procesar
for (Element elem : roundEnv.getRootElements()) {

    // COMPROBACIÓN: ELEMENTO CLASE ANOTADO CON EL TIPO ANOTACIÓN A PROCESAR

    if ((elem.getKind() != ElementKind.CLASS)
        || (elem.getAnnotation(AnotacionEjemploTypeVisitor.class) == null)) {

        // si el elemento no es una clase o no está anotado
        // con el tipo anotación a procesar -> lo ignoramos
        continue;
    }

    // down-cast a elemento de tipo (sabemos además que es una clase)
    TypeElement claseElem = (TypeElement) elem;

    // PROCESAMIENTO DEL ELEMENTO ANOTADO

    // IMPLEMENTACIÓN VISITOR TypeKindVisitor7

    // definimos el visitor que queremos utilizar
    TypeVisitor<Estadisticas, Estadisticas> typeKindVisitor7 = null;
    typeKindVisitor7 = new TypeKindVisitor7<Estadisticas, Estadisticas>() {

        @Override
        public Estadisticas visitPrimitive(
            PrimitiveType tipoPrimitivoType, Estadisticas estadisticas) {

            // CONTEO DE CAMPO PRIMITIVO

            // sumamos uno al contador de campos de tipo primitivo
            estadisticas.totalCamposPrimitivosYStrings++;

            // CONTEO ADICIONAL POR TIPO PRIMITIVO

            // llamamos al método específico de cada tipo primitivo: ¿cómo?
            // llamando a la implementación por defecto de la superclase,
            // que hace justo eso: delegar al método específico según el tipo
            super.visitPrimitive(tipoPrimitivoType, estadisticas);

            // devolvemos como resultado el parámetro
            // para obtener los resultados de las estadísticas
            return estadisticas;
        }

        @Override
        public Estadisticas visitPrimitiveAsBoolean(
            PrimitiveType tipoPrimitivoType, Estadisticas estadisticas) {
            estadisticas.totalCamposBoolean++;
            return estadisticas;
        }

        // ... redefinición de los métodos correspondientes a los demás tipos primitivos ...
        // ... visitPrimitiveAsByte, visitPrimitiveAsChar, ...
        // ... visitPrimitiveAsShort, visitPrimitiveAsInt, visitPrimitiveAsLong, ...
        // ... visitPrimitiveAsFloat y visitPrimitiveAsDouble ...
    }
}
```

```

@Override
public Estadisticas visitDeclared(
    DeclaredType tipoDeclaredType, Estadisticas estadisticas) {

    // CONTEO DE CAMPO STRING (SI PROCEDE)

    // incrementamos el total si el campo es de tipo String
    if (tipoDeclaredType.toString().equals("java.lang.String")) {

        // tipo String -> actualizamos los totales
        estadisticas.totalCamposPrimitivosYStrings++;
        estadisticas.totalCamposString++;
    }

    // devolvemos como resultado el parámetro
    // para obtener los resultados de las estadísticas
    return estadisticas;
}

// CÁLCULO DE ESTADÍSTICAS

// pasamos el visitor a los tipos de los campos del
// elemento clase anotado que tenemos que procesar

// campos de la clase
List<VariableElement> campos =
    ElementFilter.fieldsIn(claseElem.getEnclosedElements());

// inicializamos un objeto de resultados estadísticos
Estadisticas estadisticas = new Estadisticas();

// pasamos el visitor a los tipos de todos los campos
for (VariableElement campo : campos) {

    // tipo del campo
    TypeMirror tipoCampo = campo.asType();

    // pasamos el visitor al tipo del campo
    tipoCampo.accept(typeKindVisitor7, estadisticas);
}

// cuando terminamos, presentamos los resultados
this.messager.printMessage(Kind.NOTE,
    "estadisticas clase " + claseElem.getSimpleName() + ":\n" +
    "Campos de tipo primitivo y String: " + estadisticas.totalCamposPrimitivosYStrings + " (TOTAL)\n" +
    "Campos de tipo primitivo boolean: " + estadisticas.totalCamposBoolean + "\n" +
    "Campos de tipo primitivo byte: " + estadisticas.totalCamposByte + "\n" +
    "Campos de tipo primitivo char: " + estadisticas.totalCamposChar + "\n" +
    "Campos de tipo primitivo short: " + estadisticas.totalCamposShort + "\n" +
    "Campos de tipo primitivo int: " + estadisticas.totalCamposInt + "\n" +
    "Campos de tipo primitivo long: " + estadisticas.totalCamposLong + "\n" +
    "Campos de tipo primitivo float: " + estadisticas.totalCamposFloat + "\n" +
    "Campos de tipo primitivo double: " + estadisticas.totalCamposDouble + "\n" +
    "Campos de tipo primitivo String: " + estadisticas.totalCamposString);
}

} // for Element

```

Y este código imprime por la salida del compilador unos resultados idénticos al anterior. El hecho de que haya que redefinir los métodos específicos de todos los tipos primitivos (métodos **visitPrimitiveAsX**) puede hacer que esta solución salga muy aparatoso, con lo cual podría pensarse en dejar el método **visitPrimitive** como en la primera implementación, mucho más compacta, aunque fuera con un **if** múltiple, pero la clase interna **Estadisticas** sí que es a todas luces una mejora que hace que el código quede mucho más limpio.

15.3.4.- javac.

15.3.4.1.- Funcionamiento de javac.

Invocación de javac

javac, el compilador Java, inicia por defecto el procesamiento de anotaciones cuando se realiza una invocación. Típicamente se suelen indicar las opciones de compilación más algunas opciones adicionales específicas del procesamiento de anotaciones, como el procesador o procesadores de anotaciones a utilizar o las rutas donde buscar procesadores de anotaciones (en el proceso de “descubrimiento” / “discovery”), el directorio de salida de ficheros generados, los ficheros de código fuente a procesar, etcétera.

Sin entrar en detalles (para lo cual puede consultarse el subapartado “Opciones de línea de comando de **javac**”), supongamos que iniciamos el procesamiento de anotaciones con:

```
javac -cp CLASSPATH .\src\paquete\ClaseAnotada.java
```

En la invocación se indica el classpath para la compilación, así como los **ficheros de código fuente sobre los que se desea realizar el procesamiento de anotaciones**.

Ronda 0 de procesamiento

Al iniciarse el procesamiento, **javac** entra en la “ronda 0” de procesamiento, en donde sólo se llevará a cabo el descubrimiento de los procesadores de anotaciones disponibles dentro de la ruta de búsqueda para la que se haya determinado que hay que buscar.

Descubrimiento de procesadores de anotaciones

javac lleva a cabo el denominado el proceso de descubrimiento (en inglés: *discovery*) de los procesadores de anotación disponibles. Hay varias reglas que rigen este proceso, pero básicamente se trata de buscar ficheros **META-INF/services/javax.annotation.processing.Processor**, que contienen los nombres canónicos de las clases de los procesadores, recorriendo toda la “ruta de búsqueda” que se determine.



La “ruta de búsqueda” que se establezca finalmente dependerá de una serie de reglas sobre las opciones la línea de comandos establecidas en la invocación de **javac**:

Descubrimiento de procesadores de anotaciones – Ruta de búsqueda	
Opción	Procedimiento de Descubrimiento de factorías
Ninguna (por defecto)	Ruta de búsqueda del descubrimiento de procesadores: CLASSPATH. Ficheros META-INF/services/javax.annotation.processing.Processor
-processorpath PROCESSORPATH	Ruta de búsqueda del descubrimiento de procesadores: PROCESSORPATH. Ficheros META-INF/services/javax.annotation.processing.Processor
-processor Clase1,Clase2,...	Especificando los nombres canónicos (complementariamente cualificados) de las clases que implementan los procesadores de anotaciones a considerar. No se realiza el procedimiento de descubrimiento de procesadores. Se toman únicamente los procesadores especificados como los descubiertos.

El descubrimiento de procesadores se hace vía [Service Provider Interface](#) (SPI) leyendo los ficheros `META-INF/services/javax.annotation.processing.Processor` de las rutas de búsqueda (`CLASSPATH` o `PROCESSORPATH`, según sea el caso), así como esos mismos ficheros dentro de los ficheros `.jar` especificados por esas rutas de búsqueda. Recordemos que una ruta de búsqueda puede ser una lista de directorios, pero también se pueden incluir ficheros `.jar`, que tienen su propia estructura de ficheros y dentro de los cuales también se buscará.

Los ficheros `META-INF/services/javax.annotation.processing.Processor` que se vayan encontrando durante el proceso de descubrimiento, deberán tener una línea por cada procesador que implemente la interfaz `javax.annotation.processing.Processor`. La línea simplemente indicará el nombre canónico de la clase del procesador. El siguiente ejemplo muestra un fichero que contiene los nombres canónicos de tres procesadores, uno por línea:

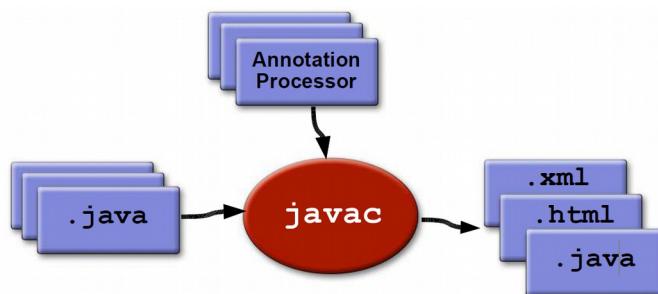
Fichero `META-INF/services/javax.annotation.processing.Processor`

```
anotaciones.procesamiento.java67.pruebas.elementosYTipos.AnotacionElementosYTiposProcessor
anotaciones.procesamiento.java67.pruebas.mirrors.AnotacionElementosClaseProcessor
anotaciones.procesamiento.java67.pruebas.visitor.AnotacionEjemploTypeKindVisitorProcessor
```

IMPORTANTE: Como se apuntaba en la tabla anterior, **en caso de utilizarse la opción `-processor Clase1,Clase2,...,ClaseN`, no se realizará el proceso de descubrimiento de procesadores** y se tomarán como únicos procesadores disponibles para el procesamiento los directamente señalados por dicha opción. En el caso del ejemplo, los dados por `Clase1, Clase2, hasta ClaseN`.

Ronda 1 de procesamiento de anotaciones

Una vez realizado el descubrimiento de procesadores, se inicia la “ronda 1” o “primera ronda” de procesamiento, cuya entrada serán los ficheros especificados en la invocación inicial:



Sobre los ficheros que se pasan como entrada a `javac` se hará:

- 1) Un análisis para la **búsqueda de tipos anotación a procesar** que contienen.
- 2) Un **emparejamiento de procesadores de anotaciones ↔ tipos anotación**.
- 2 & 3) El **procesamiento de anotaciones** como tal, mientras se realiza el emparejamiento.

Búsqueda de tipos anotación a procesar

Al inicio de cada ronda de procesamiento, `javac` realiza una **búsqueda de tipos anotación** presentes en el código fuente que se va a procesar para saber los tipos anotación que tendrá que procesar mediante procesamiento de anotaciones. En este caso, se analizará el texto del código fuente del fichero `ClaseAnotada.java` para encontrar qué anotaciones contiene. Supongamos que `javac` encuentra varias anotaciones del tipo anotación ficticio `@AnotacionEjemplo`.



Emparejamiento de procesadores de anotaciones ↔ tipos anotación

Una vez que `javac` ha determinado qué tipos anotación tiene que procesar, conociendo de la “ronda 0” con qué procesadores cuenta, realiza un emparejamiento entre procesadores y tipos anotación a procesar. Para ello, sigue el siguiente **procedimiento de emparejamiento**:

- 1) `javac` solicita a cada procesador una lista de los tipos anotación que procesa.
- 2) `javac` asignará el tipo anotación al procesador si procesa dicho tipo anotación.
- 3) `javac` invoca al procesador asignado y, si este “reclama” (del inglés: “claim”) dicho tipo anotación, el tipo saldrá de la lista de tipos anotación pendientes. Si el procesador no “reclama” el tipo anotación para sí mismo, `javac` buscará más procesadores aplicables al tipo anotación.
- 4) Este proceso continuará hasta que todos los tipos anotación hayan sido “reclamados” (y, por tanto, la lista de tipos anotación pendientes es vacía) o no haya más procesadores aplicables.

Interacción de `javac` con los procesadores de anotaciones

Cuando `javac` necesita interactuar con un procesador para interrogarlo, por ejemplo, sobre los tipos anotación que soporta para el emparejamiento, sigue los siguientes pasos:

- 1) Si no ha sido instanciado con anterioridad, `javac` creará una instancia de dicho procesador llamando al constructor sin argumentos de la clase que implementa el procesador.
- 2) `javac` llama al método `init` del procesador pasándole un `ProcessingEnvironment`.
- 3) `javac` llama a todos los métodos que le permiten recabar información acerca del procesador: `getSupportedAnnotationTypes`, `getSupportedOptions`, y `getSupportedSourceVersion`.

Procesamiento de anotaciones

Una vez que `javac` logra emparejar un procesador con un tipo anotación, para que este lleve a cabo el procesamiento correspondiente, llama al método `process` del procesador:

```
boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)
```

El método `process` recibe como parámetros un conjunto de tipos anotación a procesar (**CUIDADO**: los tipos anotación, no las anotaciones en sí) y un `RoundEnvironment`, “entorno de ronda”, con la información necesaria para implementar el procesamiento en la ronda actual.

El **valor de retorno boolean del método process** indica si el procesador “reclama” para sí mismo los tipos de anotación pasados como argumento. En caso de que los reclame, estos tipos anotación salen de la lista de tipos anotación pendientes y ya no podrán ser reclamados por otro procesador de anotaciones dentro de la presente ronda. Normalmente, dado que los procesadores de anotaciones no deberían interferir unos con otros, se recomienda devolver `false` para no impedir la ejecución de otros procesadores de anotaciones.

En caso de “procesadores universales”, cuyo tipo anotación soportado es “`*`” que significa “todos los tipos anotación”, cuando `javac` llama al `process` de un “procesador universal” el conjunto de tipos anotación se pasa vacío. En ese caso, además, si el procesador devolviera `true`, reclamaría todos los tipos anotación para sí mismo y evitaría que se ejecutara cualquier otro procesador de anotaciones. En este caso, en la documentación oficial se recomienda que se devuelva `false` para evitar impedir que se ejecuten otros procesadores, sean universales o no.

El procesamiento de anotaciones dentro de una ronda continuará mientras queden tipos anotación pendientes que no hayan sido emparejados. Si todos los procesadores fueran “reclamando” sus tipos anotación, la lista llegaría a vaciarse y ya no se seguirían buscando más procesadores. Si no, el procesamiento terminará, aunque queden tipos anotación pendientes de procesar, cuando se ejecuten todos los procesadores disponibles.

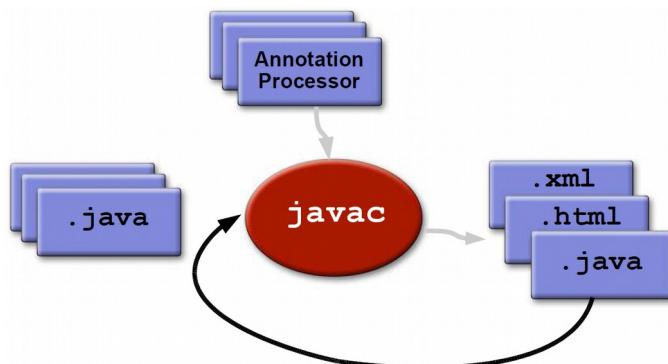
Cuando todos y cada uno de los procesadores de anotaciones descubiertos y emparejados hayan finalizado la ejecución de su método **process**, terminará la “ronda 1” o “primera ronda” del procesamiento de anotaciones. Y, si cualquier procesador ha generado nuevos ficheros de código fuente durante la ronda, incluso se producirá una “ronda adicional”.

Rondas adicionales de procesamiento de anotaciones (si procede)

El concepto de “ronda” (del inglés: “round”) surge debido al hecho de que los procesadores de anotaciones pueden generar nuevo código fuente. Y dicho nuevo código fuente generado ¡¡ipodría a su vez contener anotaciones que es necesario procesar!!!

Es decir, que **mientras que haya procesadores de código que generen nuevo código fuente, el procesamiento de anotaciones no puede finalizar, debe continuar con una nueva “ronda adicional de procesamiento”.**

Si se hace necesaria una ronda adicional de procesamiento debido a que algún procesador de anotaciones ha generado nuevo código fuente, **javac** pasará a realizar de nuevo los mismos pasos que se llevaron a cabo durante la primera ronda, pero esta vez lo que se tomará como entrada de código fuente a procesar será el nuevo código fuente recién generado en la ronda anterior, como se ilustra en el esquema:



javac realizará de nuevo los mismos procesos que en la primera ronda:

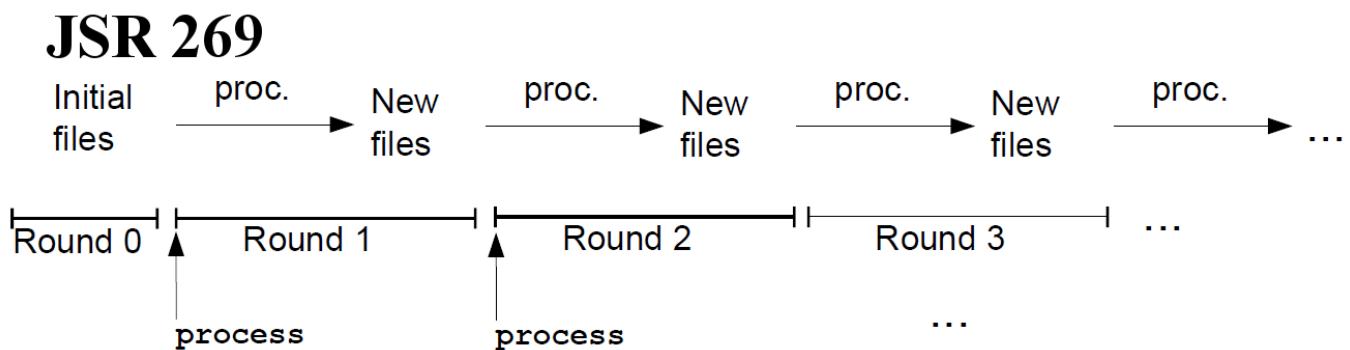
- 1) Un análisis para la búsqueda de tipos anotación a procesar que contienen.
- 2) Un emparejamiento de procesadores de anotaciones ↔ tipos anotación.
- 2 & 3) El procesamiento de anotaciones como tal, mientras se realiza el emparejamiento.

IMPORTANTE: En las rondas subsiguientes a la primera no se instanciarán de nuevo los procesadores que ya hubieran sido instanciados con anterioridad. **Crear una instancia, llamar a `init` y a los métodos informativos sólo se hace una vez por procesador** dentro de una misma ejecución. Además, ocurre otro comportamiento bastante particular: **en cada ronda subsiguiente siempre se ejecutarán de nuevo todos los procesadores ejecutados con anterioridad, aunque no tengan anotaciones que procesar.**

Procesamiento de anotaciones: un proceso recursivo

El procesamiento de anotaciones puede convertirse en un **proceso recursivo** con un número desconocido de “rondas de procesamiento”, antes de llegar a terminar cuando se alcanza el final de una ronda sin que se haya generado nuevo código fuente.

Siendo así, el esquema temporal en forma de cronograma que sigue una ejecución del procesamiento de anotaciones JSR 269 en `javac` con múltiples rondas de procesamiento (incluyendo sub-etapas: buscar anotaciones, emparejamiento procesadores ↔ tipos anotación, procesamiento con `process` y generación de nuevos ficheros) tendría el siguiente aspecto:



Las rondas de procesamiento adicionales pueden ser ninguna, una o incluso, dado que es un proceso recursivo, podría generarse un “bucle infinito de procesamiento de anotaciones” si ocurriera que el diseño de los procesadores provocase que siempre se genere nuevo código fuente en cada nueva ronda. Esto sería así porque, en ese caso, no se estaría respetando el “caso base” de la recursión (no generar nuevos ficheros) y podríamos convertir el proceso en una recursión “no convergente” o “divergente”, es decir, que no se acerque al caso base recursivo y que, por tanto, no llegue a terminar jamás.

Así pues, podría darse un “bucle infinito de procesamiento de anotaciones”, por ejemplo, si diseñáramos un procesador de anotaciones que siempre generase código fuente nuevo (**GeneradorEternoCodigoFuenteProcessor**). O también cuando dos o más procesadores de anotaciones generasen tipos anotación relacionados y se creara un lazo entre ellos:

Entrada: ClaseA anotada con una anotación @GeneraA

Ronda 1: GeneradorClaseAProcessor → genera nueva ClaseA1 anotada con @GeneraB

Ronda 2: GeneradorClaseBProcessor → genera nueva ClaseB1 anotada con @GeneraA

Ronda 3: GeneradorClaseAProcessor → genera nueva ClaseA2 anotada con @GeneraB

Ronda 4: GeneradorClaseBProcessor → genera nueva ClaseB2 anotada con @GeneraA

... *ad infinitum* ... (o más bien exactamente hasta que el entorno de ejecución arroje un **OutOfMemoryError**)

Ronda final del procesamiento de anotaciones

Una vez se haya completado una ronda de procesamiento sin haber generado nuevo código fuente, se da por concluida la “ronda final” y, por ende, se da por concluido todo lo que es el procesamiento de anotaciones como tal.

Terminado el procesamiento, por defecto, **javac** realizará la compilación de todos los ficheros de código fuente, tanto los que había originalmente antes del procesamiento, como todos los nuevos ficheros de código que se hayan podido generar a lo largo de todas las rondas de procesamiento de anotaciones.

NOTA: La compilación final de **javac** puede no darse si se especifica la opción **-proc:only** en la línea de comandos (para que el compilador sólo haga el procesamiento de anotaciones en exclusiva y no la compilación). Esta opción, como veremos más adelante, puede tener mucho interés a la hora de depurar errores cometidos en los ficheros generados por los procesadores.

Código de retorno de **javac**

El código de retorno (en inglés “return code”, “exit code” o “exit status”) de **javac** puede ser de mucha importancia en caso de programar scripts de procesos de compilación o de construcción de aplicaciones. Los códigos de retorno posibles de **javac** son los siguientes:

Códigos de retorno de javac		
Código de retorno	Nombre interno	Descripción
0	EXIT_OK	Compilación completada sin errores.
1	EXIT_ERROR	Compilación completada con errores.
2	EXIT_CMDERR	Error en los argumentos de línea de comandos.
3	EXIT_SYSERR	Error de sistema o agotamiento de los recursos disponibles.
4	EXIT_ABNORMAL	Terminación anormal del compilador debido a algún fallo interno.

javac finalizará siempre devolviendo alguno de los códigos de retorno indicados en la tabla anterior. Devolverá un código de retorno **0** si no se produjeron errores de ningún tipo durante su ejecución, ya sea durante el procesamiento de anotaciones o en la compilación propiamente dicha. Y devolverá un código de retorno distinto de **0** si ocurre cualquier tipo de error en cualquier momento a lo largo de alguna de las dos etapas.

15.3.4.2.- Opciones de línea de comandos de `javac`.

En este apartado se recoge una referencia de las opciones de línea de comando de `javac` de interés exclusivo para el procesamiento de anotaciones. Por supuesto, hay que recordar que `javac`, como compilador, dispone de muchas otras opciones adicionales no relacionadas con el procesamiento (úse `javac -help` para obtener información de las opciones disponibles).

Opciones de línea de comandos de <code>javac</code>	
Opción	Descripción
<code>-classpath CLASSPATH</code>	Ruta de los ficheros clases de usuario de donde cargar las definiciones de las clases a emplear tanto en el procesamiento de anotaciones como al compilar. Puede incluir tanto directorios como ficheros <code>.jar</code> . De hecho, normalmente el <code>CLASSPATH</code> incluye la ruta base de los ficheros de clase compilados del usuario, así como tantas librerías <code>.jar</code> sean necesarias para sus dependencias. Si no se especifica <code>-processorpath</code> , se tomará también el <code>CLASSPATH</code> como la ruta de búsqueda para el proceso de descubrimiento de procesadores en los ficheros correspondientes: <code>META-INF/services/javax.annotation.processing.Processor</code>
<code>-s DIR_FUENTES_GENERADAS</code>	Directorio raíz donde colocar los ficheros de código fuente generados durante el procesamiento de anotaciones. Cada fichero se guardará en el subdirectorio que le corresponda según su respectivo paquete.
<code>-d DIR_DESTINO_CLASES</code>	Directorio raíz donde colocar los ficheros de clase generados al compilar todos los ficheros de código fuente existentes tras la última ronda de procesamiento (ficheros originales más los generados). Cada fichero de clase se guardará en el subdirectorio que le corresponda según su respectivo paquete.
<code>-source VERSION_JAVA</code>	Especifica que se quiere compatibilidad a nivel de código fuente con la versión de Java indicada. Por defecto, se tomará el nivel de código de la JDK a la que pertenezca la implementación de <code>javac</code> con la que estemos trabajando.
<code>-processorpath PROCPATH</code>	Ruta de los ficheros clases de usuario de donde realizar el descubrimiento de procesadores de anotaciones. Puede incluir tanto directorios como ficheros <code>.jar</code> . Para dicha ruta de búsqueda, se buscarán procesadores en los ficheros correspondientes: <code>META-INF/services/javax.annotation.processing.Processor</code>
<code>-processor Clase[,Clase]</code>	Especifica que no se realice el descubrimiento de procesadores, y que se tomen en consideración los procesadores indicados por la clase o clases especificadas. Los nombres de las clases de los procesadores se dan como nombres canónicos.
<code>-proc:none</code>	<code>javac</code> no hace el procesamiento de anotaciones. Sólo compila el código.
<code>-proc:only</code>	<code>javac</code> sólo hace el procesamiento de anotaciones. No se compila el código.
<code>-Aclave[=valor]</code>	Opciones para los procesadores de anotaciones.
<code>-implicit:{class,none}</code>	Generación de ficheros de clase para ficheros fuente cargados implicitamente. <code>-implicit:class</code> (por defecto) para la generación de dichos ficheros de clase e <code>-implicit:none</code> para suprimir dicha generación. Si la opción <code>-implicit</code> no se especifica, el compilador emitirá un warning en caso de generar ficheros de clase implícitos cuando también se esté haciendo procesamiento de anotaciones.
<code>-version</code>	Imprime información de la versión de <code>javac</code> . Por ejemplo: <code>javac 1.8.0_05</code> .
<code>-help</code>	Imprime una sinopsis de las opciones estándar de <code>javac</code> .
<code>-X</code>	Imprime una sinopsis de las opciones no estándar de <code>javac</code> .
<code>-XprintProcessorInfo</code>	Información de qué tipos anotación procesa un procesador. Para depuración.
<code>-XprintRounds</code>	Información acerca de las rondas de procesamiento. Sirve para depuración.
<code>-Xlint:processing</code> (sólo desde Java SE 7)	Activa la emisión de warnings por parte del compilador que tengan que ver con procesamiento de anotaciones. Por ejemplo: que se emita un warning que liste los tipos anotación no reclamadas por ningún procesador de anotaciones. Por ejemplo, supongamos que el tipo anotación Anot no es procesado por ninguno de los procesadores descubiertos por <code>javac</code> . Se emitiría el siguiente warning: <code>warning: [processing] No processor claimed any of these annotations: Anot</code> Sirve para depuración y puede ser interesante activarlo como complemento adicional a la información proporcionada por las demás opciones no estándar.

Sintaxis general de javac

```
javac [-classpath CLASSPATH]
      [-s DIR_FUENTES_GENERADAS] [-d DIR_DESTINO_CLASES]
      [-source VERSION_JAVA]
      [-processorpath PROCESSORPATH] [-processor Clase[,Clase]]
      [-proc:{none,only}]
      [-Aclave[=valor] ...]
      [-implicit:{class,none}]
      [-version] [-help]
      [-X] [-XprintProcessorInfo] [-XprintRounds] [-Xlint:processing]
      [otras opciones javac] FICHEROS_FUENTE [@FICHEROS_INLINE]
```

El orden es el de la referencia de la página anterior, aunque es posible especificar las opciones en cualquier orden, siempre que aparezcan antes de los **FICHEROS_FUENTE**.

Todas las opciones están entre corchetes [] porque el único parámetro realmente obligatorio son los **FICHEROS_FUENTE** a procesar. Todas las otras opciones no son obligatorias porque, o tienen valores por defecto, o realizan acciones aisladas que no hay por qué especificar.

javac también permite especificar listas de ficheros fuente o de opciones a través de **@FICHEROS_INLINE**, esto es, ficheros de texto con una línea por fichero u opción.

Recomendaciones sobre el establecimiento de directorios para javac

Siempre que sea posible, se considera higiénico separar los diferentes directorios y rutas de búsqueda que entran en juego a lo largo de todo el proceso llevado a cabo por **javac**, de forma que las entradas y las salidas al compilador estén claramente separadas:

Rutas de directorios de ficheros para la entrada y salida de javac		
Opción	Dirección	Valor recomendado
-cp CLASSPATH	ENTRADA	Directorio raíz de las clases compiladas de la aplicación, así como de todos los ficheros .jar necesarios para la compilación.
-s DIR_FUENTES_GENERADAS	SALIDA	Suele ser un directorio llamado generated_src o similar.
-d DIR_DESTINO_CLASES	SALIDA	Suele ser un directorio llamado normalmente classes o bin .

Si se mezclan varios tipos de ficheros en el directorio de destino de clases, ello podría provocar problemas adicionales si los ficheros originales se encontraran bajo algún tipo de control de versiones de código fuente. Los ficheros derivados no deberían nunca de estar bajo ningún tipo de control de versiones.

15.3.4.3.- Invocación de javac.

Hay muchas formas de invocar a **javac** para lanzar el procesamiento de anotaciones. En función de nuestras necesidades o de cómo se haya organizado el proceso de desarrollo y compilación de una aplicación concreta, puede venirnos mejor un método u otro.

Se ha tratado de ser exhaustivos y se explican todos los métodos posibles de invocación de **javac**. Aunque algunos de ellos carecerán de interés en general, como las modalidades de invocaciones programáticas, pueden ser interesantes para el desarrollo de front-ends, wrappers o plugins que tengan que invocar el procesamiento de anotaciones y/o la compilación.

De mayor utilidad práctica y de un uso mucho más extendido es la invocación a través de [Apache Maven](#), el sistema de administración de dependencias (“dependency management”) y construcción (“build”) de aplicaciones más extendido en la actualidad en la comunidad Java.

Finalmente, se incluye una referencia al proceso de configuración de procesamiento de anotaciones en el popular IDE [Eclipse](#). Para saber del procedimiento de configuración necesario para el procesamiento de anotaciones en otros IDEs como [NetBeans](#), [IntelliJ IDEA](#) o cualquier otro de los [IDEs Java](#) disponibles, por favor, acuda a la documentación oficial de los mismos.

15.3.4.3.1.- Invocación de javac mediante línea de comandos.

La invocación directa de **javac** por línea de comandos, aunque es la forma más básica y estándar, es poco práctica a la hora de sobrellevar un desarrollo de aplicaciones intenso. El medio más básico para invocarlo es desde el intérprete de comandos de nuestro sistema operativo: la ventana de comandos de Windows, una consola o terminal de GNU/Linux, etc. Obviamente, para no tener que estar escribiendo continuamente el comando necesario, se definirán, como poco, unos ficheros de script que nos permitan lanzar **javac** de forma cómoda.

Ejemplo: Invocación de proc. anotaciones con javac desde Windows

```
javac -version -source 7 -proc:only -Xlint:-options ^
 -cp .\target\classes ^
 -s .\src\main\java-generated ^
 -Aopcion.valor.string="¡Hola, mundo!" -Aopcion.valor.integer=12345 ^
 -processor anotaciones.procesamiento.java67.pruebas.visitor.typeVisitor.VisitorProcessor ^
 .\src\main\java\anotaciones\procesamiento\java67\pruebas\visitor?typeVisitor\*.java
```

NOTA 1: El acento circunflejo ^ se usa como marca de salto de línea para usar esta expresión en un fichero de script de proceso por lotes (extensión **.bat**) en Windows. La línea siguiente a la del acento circunflejo ^ no puede empezar en la posición 0, por lo que se recomienda colocar al menos un espacio, como se ve en el ejemplo en las líneas 2 y siguientes. Esto nos permite crear unos scripts escalonados en varias líneas mucho más sencillos de leer y, al facilitar que las líneas sean menos largas, sin tener que desplazarnos tanto con el scroll lateral a izquierda y derecha.

NOTA 2: El directorio donde se ubica **javac**, normalmente **JDK_HOME\bin**, debe estar en la ruta de búsqueda de ejecutables del sistema operativo para que el comando funcione tal cual está escrito. En caso de que esto no sea así, puede incluirse la ruta de **javac** en dicha ruta de búsqueda de ejecutables (en Windows: incluir **%JDK_HOME%\bin** como parte de la variable de entorno **Path**) o simplemente poner la ruta completa al directorio del ejecutable, por ejemplo:

```
C:\Software\Desarrollo\Java\JSDK\JavaSE7\bin\javac -version ... resto del comando ...
```

15.3.4.3.2.- Invocación de javac programática como proceso.

Una invocación de `javac` por línea de comandos puede recrearse programáticamente usando `ProcessBuilder` y `Process` del paquete `java.lang` de la forma siguiente:

```
protected static void lanzarJavacProgramaticamenteComoProceso() throws IOException {

    // ruta al directorio de la JDK donde está el ejecutable "javac.exe"
    String javacRutaEjecutable =
        "C:\\Software\\Desarrollo\\Java\\JSDK\\JavaSE7\\bin";

    // para que las rutas relativas de la llamada original se resuelvan a las correctas
    // (otra opción es pasarle siempre las rutas absolutas resueltas correspondientes)
    String rutaBase =
        "D:\\PFC\\Contenido\\3-Desarrollo\\EclipseWorkspace\\anotaciones-proc-java67";

    String[] argumentos =
    {
        javacRutaEjecutable + "\\javac.exe",
        "-version", "-source", "7",
        "-proc:only", "-Xlint:-options",
        "-cp", ".\\target\\classes",
        "-s", ".\\src\\main\\java-generated",
        "-Aopcion.valor.string=Hola, mundo!", "-Aopcion.valor.integer=12345",
        "-processor", "anotaciones.procesamiento.java67.pruebas.visitor.typeVisitor.AnotacionEjemploTypeKindVisitorProcessor",
        ".\\src\\main\\java\\anotaciones\\procesamiento\\java67\\pruebas\\visitor\\typeVisitor\\*.java"
    };

    // construimos el proceso en base a sus argumentos
    ProcessBuilder processBuilder = new ProcessBuilder(Arrays.asList(argumentos));

    // establecemos el directorio de trabajo como la ruta base donde está el proyecto
    processBuilder.directory(new File(rutaBase));

    // IMPORTANTE: redirigir la salida de error a la salida estándar para poder verla
    processBuilder.redirectErrorStream(true);

    // arrancamos el proceso
    final Process proceso = processBuilder.start();

    // imprimimos por consola la salida de la ejecución del proceso
    BufferedReader br = new BufferedReader(new InputStreamReader(proceso.getInputStream()));
    String lineaSalidaProceso;
    while ((lineaSalidaProceso = br.readLine()) != null) {
        System.out.println(lineaSalidaProceso);
    }

    // imprimimos el código de retorno
    System.out.println("Código de retorno: " + proceso.exitValue());
}
}
```

Consideraciones sobre el uso de `java.lang.Process` y `java.lang.ProcessBuilder`

- Las rutas relativas al directorio actual “.” no funcionan sin `rutaBase` como directorio.
- Las opciones que dejan espacios en blanco intermedios con su valor (como `-cp` o `-s`), han de partirse en 2 para que el parser de `javac` las coja. Si no da un error de `invalid flag`.
- Al usar `String`'s ya no habrá que utilizar comillas “” para marcar el inicio y el final de los valores que, por ejemplo, contengan espacios, pero sí **será necesario escapar la \ con **.

15.3.4.3.3.- Invocación de javac programática vía Java Compiler API.

La modalidad de invocación programática de **javac** más oficializada que existe es a través de la **Java Compiler API (JSR 199)**. Esta API permite abre muchas posibilidades para la integración de la administración de la compilación dentro de las propias aplicaciones Java.

Queriendo realizar la misma invocación ejemplo de **javac** por línea de comandos, dicha llamada se puede invocar programáticamente en la Java Compiler API de 2 formas:

(1) “**Método clásico**”: se le pasan directamente todos los argumentos en un **String[]** al método **run** del compilador y **se obtiene como resultado el clásico código de retorno**.

(2) “**Método nuevo**”: haciendo uso de las nuevas clases de la Java Compiler API que permiten crear “tareas de compilación”, se ejecutan con su método **call** y **se obtiene como resultado un booleano que nos indica si la compilación ha finalizado sin errores**.

El siguiente código de ejemplo contiene al principio una parte común y después un bloque específico para cada método de invocación, primero el “**método clásico**” y luego el “**método nuevo**”. Así pues, la compilación se realizará 2 veces. Esto es, por supuesto, debido a que dicho código es meramente ilustrativo para poner de relieve el uso de ambos métodos. Los usuarios de la Java Compiler API tendrán que decidir cuál entre ambos métodos utilizar.

```
protected static void lanzarJavacMedianteCompilerAPI() {

    // COMPILADOR Y SISTEMA DE FICHEROS

    // obtenemos una instancia del compilador Java
    JavaCompiler compilador = ToolProvider.getSystemJavaCompiler();
    System.out.println("compilador = " + compilador);

    // instancia del manejador de ficheros
    StandardJavaFileManager javaFileManager =
        compilador.getStandardFileManager(
            null, // diagnosticListener
            null, // locale
            null); // charset

    // OPCIONES DE COMPILACIÓN
    // (CLASSPATH, DIRECTORIOS DE SALIDA, OPCIONES DE PROCESADORES)
    // IMPORTANTE: Algunas opciones que sí se soportan desde línea de comandos,
    // como -version, NO están soportadas si se pasan a través de la Compiler API,
    // y al intentar pasarlas, el compilador se quedará de una "invalid flag".

    // opciones de compilación
    String[] opcionesCompilacion = new String[]
    {
        "-source", "7", "-proc:only", "-Xlint:-options",
        "-cp", ".\\target\\classes",
        "-s", ".\\src\\main\\java-generated",
        "-d", ".\\target\\classes",
        "-Aopcion.valor.string=;Hola, mundo!", "-Aopcion.valor.integer=12345",
        "-processor",
        "anotaciones.procesamiento.java67.pruebas.visitor.typeVisitor.AnotacionEjemploTypeKindVisitorProcessor"
    };

    // mostramos las opciones de compilación
    System.out.println("opcionesCompilacion = " + Arrays.toString(opcionesCompilacion));
}
```

```

// DIRECTORIOS Y FICHEROS A COMPILAR

// IMPORTANTE: A diferencia de la línea de comandos, que puede llegar a aceptar
// un comodín * en un directorio tal que así:
// .\src\main\java\anotaciones\procesamiento\java67\pruebas\visitor\typeVisitor\*.java
// la Java Compiler API no soporta este mecanismo, por lo que hay que construir
// la lista de ficheros a compilar listando los ficheros de tales directorios.

// lista de directorios a compilar (se listarán sus ficheros respectivos)
ArrayList<File> directoriosACompilar = new ArrayList<File>();

// lista final de ficheros a compilar
ArrayList<File> ficherosACompilar = new ArrayList<File>();

// filtro para nombres de ficheros que terminen en .java
FilenameFilter filtroFicherosJava = new FilenameFilter() {

    @Override
    public boolean accept(File dir, String name) {

        // true si el nombre del fichero termina en .java
        return name.endsWith(".java");
    }
};

// CONSTRUCCIÓN DE LA LISTA DE FICHEROS A COMPILAR

// directorio(s) a compilar
directoriosACompilar.add(
    new File("./\src\main\java\anotaciones\procesamiento\" +
        + "java67\pruebas\visitor\typeVisitor"));

// listamos los ficheros de los directorios que queremos compilar
// y los añadimos a la lista final de ficheros a compilar
for (File directorioACompilar : directoriosACompilar) {

    // listamos los ficheros .java del directorio a compilar
    File[] ficherosJavaDirectorioACompilar =
        directorioACompilar.listFiles(filtroFicherosJava);

    // añadimos los ficheros del directorio a la lista de ficheros a compilar
    ficherosACompilar.addAll(Arrays.asList(ficherosJavaDirectorioACompilar));
}

// mostramos la lista final de ficheros a compilar
System.out.println("ficherosACompilar (" + ficherosACompilar.size() + ") = " + ficherosACompilar);

// *** MÉTODO CLÁSICO: INVOCACIÓN AL COMPILADOR CON CÓDIGO DE RETORNO **

// este es el método clásico donde se invoca al compilador
// pasándole un array con todos los argumentos necesarios
// (opciones + lista de rutas de ficheros a compilar)
// y el compilador devuelve un código de retorno
System.out.println("compilacion: metodo clasico: iniciando...");

// argumentos = opciones de compilación + ficheros a compilar

// creamos la lista de argumentos inicialmente con las opciones
ArrayList<String> argumentosLineaComandos =
    new ArrayList<String>(Arrays.asList(opcionesCompilacion));

// añadimos los ficheros a compilar
for (File ficheroACompilar : ficherosACompilar) {

    // añadimos los ficheros a compilar con su ruta completa
    argumentosLineaComandos.add(ficheroACompilar.getAbsolutePath());
}

```

```

// mostramos el resultado de la compilación
System.out.println("compilacion: metodo clasico: "
    + "argumentosLineaComandos = " + argumentosLineaComandos);

// pasamos la lista a un array, que es lo que requiere el método run
String[] argumentosLineaComandosArray = argumentosLineaComandos.toArray(new String[] {});

// llamamos al compilador pasándole los argumentos y recogemos el código de retorno
System.out.println("compilacion: metodo clasico: compilando...");
int codigoRetorno = compilador.run(null, // in
    null, // out
    null, // err
    argumentosLineaComandosArray); // arguments

// mostramos el resultado de la compilación
System.out.println("compilacion: metodo clasico: "
    + "codigoRetorno = " + codigoRetorno);

// *** MÉTODO NUEVO: COMPILACIÓN COMO TAREA (DEVUELVE TRUE/FALSE) **

// este nuevo método utiliza las clases y facilidades puramente nuevas
// de la Java Compiler API para modelar tareas de compilación
System.out.println("compilacion: metodo nuevo: iniciando...");

// mostramos las opciones de compilación (de nuevo, como recordatorio)
System.out.println("compilacion: metodo nuevo: "
    + "opcionesCompilacion = " + Arrays.toString(opcionesCompilacion));

// unidades de compilación (JavaFileObject's a compilar)
Iterable<? extends JavaFileObject> unidadesCompilacion =
    javaFileManager.getJavaFileObjectsFromFiles(ficherosACompilar);
System.out.println("compilacion: metodo nuevo: "
    + "unidadesCompilacion = " + unidadesCompilacion);

// creamos la tarea de compilación
CompilationTask tareaCompilacion =
    compilador.getTask(
        null, // out
        javaFileManager, // fileManager
        null, // diagnosticListener
        Arrays.asList(opcionesCompilacion), // options
        null, // classes
        unidadesCompilacion);

// NOTA: Los procesadores de anotación también pueden establecerse,
// además de con la opción -processor de línea de comandos,
// llamando al método tareaCompilacion.setProcessors(procesadores);
// No obstante, este último método requiere que los procesadores
// sean pasados como objetos instanciados no como Strings.

// ejecutamos la tarea de compilación y obtenemos el resultado
System.out.println("compilacion: metodo nuevo: compilando...");
boolean resultadoCompilacion = tareaCompilacion.call();

// mostramos el resultado de la compilación
System.out.println("compilacion: metodo nuevo: "
    + "resultadoCompilacion (sin errores) = " + resultadoCompilacion);
}

```

Consideraciones sobre el uso de la Java Compiler API

- Las opciones que dejan espacios en blanco intermedios con su valor (como **-cp** o **-factory**), así como la opción especial **-J**, habrán de partirse en 2 argumentos para que se parseen bien.
- Al usar String's ya no habrá que utilizar comillas "" para marcar el inicio y el final de los valores que, por ejemplo, contengan espacios, pero sí **será necesario escapar la \ con **.
- Las rutas relativas al directorio actual .. funcionarán sólo si la invocación se hace en el directorio adecuado. Si no, hay que (1) establecer todas las rutas como absolutas o (2) establecer la propiedad **"user.dir"** de la MV Java al directorio a usar como directorio actual con **"-J" , "-Duser.dir=" + rutaBase** entre los **argumentosLineaComandos** para el "método clásico" (nótese que está partido en 2 argumentos) o con **-Duser.dir=valor** como parámetros del comando **java** invocador en el caso del "método nuevo" (donde **PruebaCompilerAPI** es la clase que aloja el código que lanza la invocación de la Java Compiler API para el "método nuevo"):

```
java.exe ^
-Duser.dir=D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-java67\ ^
-cp .\target\classes ^
anotaciones.procesamiento.java67.pruebas.PruebaCompilerAPI
```

Java Compiler API en Eclipse: No soportada

Hay que hacer un comentario especial sobre el uso de la Java Compiler API en el entorno de desarrollo [Eclipse](#). Dicho entorno implementa su propio compilador de Java, como veremos más detalladamente en el apartado "Invocación de javac mediante Eclipse". Esto hace que la llamada **ToolProvider.getSystemJavaCompiler()** no retorne ninguna implementación de la interfaz **JavaCompiler**, si no **null**.

```
// obtenemos una instancia del compilador Java
JavaCompiler compilador = ToolProvider.getSystemJavaCompiler(); // !!!devuelve null!!!
System.out.println("compilador = " + compilador);
```

Al ejecutar este código con el compilador y la implementación del entorno de ejecución de Eclipse obtenemos la siguiente respuesta:

```
compilador = null
Exception in thread "main" java.lang.NullPointerException
    at
anotaciones.procesamiento.java67.pruebas.PruebaCompilerAPI.lanzarJavacMedianteCompilerAPI
(PruebaCompilerAPI.java:31)
    at
anotaciones.procesamiento.java67.pruebas.PruebaCompilerAPI.main(PruebaCompilerAPI.java:18)
```

Siendo así, no será posible utilizar la insfraestructura proporcionada por la Java Compiler API si se invoca dicho código dentro del entorno de ejecución de Eclipse.

La solución, por supuesto, será utilizar la implementación del compilador **javac** oficial de Oracle (implementación de referencia) o cualquier otra que sí disponga de un objeto que implemente la interfaz **JavaCompiler** al llamar a **ToolProvider.getSystemJavaCompiler()**.

15.3.4.3.4.- Invocación de `javac` mediante Maven.

Apache Maven es hoy en día el sistema de administración de dependencias (“dependency management”) y construcción (“build”) de aplicaciones más utilizado en la comunidad Java. Su estandarización del proceso de construcción y la enorme productividad que ello ofrece han hecho de Maven un estándar de facto entre la comunidad de desarrollo Java. Para más detalles sobre Maven, puede consultarse su página web oficial apache.maven.org.

Maven puede invocar a `javac` para realizar procesamiento de anotaciones utilizando cualquiera de los siguientes plugins:

- (1) **maven-processor-plugin** [<https://github.com/bsorrentino/maven-annotation-plugin>].
- (2) **maven-compiler-plugin** [<http://maven.apache.org/plugins/maven-compiler-plugin/>]

En principio, vamos a preferir la opción (1) **maven-processor-plugin**, ya que se trata de un plugin especializado en la configuración del procesadores de anotaciones, mientras que la opción (2) el **maven-compiler-plugin** se trata del plugin base de Maven para llamar a los compiladores de código, que en el caso de usar `javac` soporta procesamiento de anotaciones.

En cuanto al descubrimiento de los procesadores de anotación disponibles, será necesario que las clases de los mismos se encuentren disponibles en el classpath de compilación. Para ello, suele ser habitual especificar el archivo `.jar` que contiene las definiciones de los procesadores a ejecutar como dependencia Maven. En el caso del siguiente ejemplo no es así, porque nosotros vamos a estar modificando continuamente el procesador para depurarlo y sería, como poco, una molestia tener que estar regenerando el `.jar` a cada pequeño ajuste del código.

De hecho, en caso de estar desarrollando y depurando un procesador de anotaciones, lo mejor es especificarlo directamente para obviar el proceso de descubrimiento y que se ejecute directamente. Esta ha sido la forma habitual de trabajar que se ha seguido para el desarrollo de los ejemplos de procesamiento de anotaciones que se incluyen en este manual: usar Maven para la configuración de la invocación del procesador de anotaciones, observando la información de salida de los mismos a través de la consola del entorno Eclipse.

Si no se está desarrollando un procesador de anotaciones, si no que se es usuario de una librería que nos proporciona procesadores de anotaciones ya desarrollados, y queremos que puedan ser descubiertos, como hemos dicho, sólo habrá que añadir como dependencia dicha librería, para que sean incluidas sus clases y recursos como parte del classpath de compilación.

Procesamiento de anotaciones con maven-processor-plugin

El **maven-processor-plugin** puede utilizarse por sí sólo para invocar a **javac** para realizar procesamiento de anotaciones. No obstante, la práctica recomendada es que se use sólo para realizar el procesamiento de anotaciones especificando la opción **-proc:only** y que se deje la compilación al **maven-compiler-plugin**. Por tanto, después de todo, sí utilizaremos el **maven-compiler-plugin**, aunque sólo sea para la compilación.

Se utilizan ambos plugins para separar las diferentes fases de trabajo con el código fuente y permitir una posible configuración separada y avanzada de la compilación si esta fuera necesaria, algo para lo que el **maven-compiler-plugin** sí es la mejor opción.

En el ejemplo que se muestra a continuación se han especificado pues las ejecuciones de ambos plugins: el **maven-processor-plugin** sólo para el procesamiento especificando la opción **-proc:only**, y luego el **maven-compiler-plugin** que sólo realiza la compilación final del código generado durante el procesamiento especificando la opción **-proc:none**.

Para usar el **maven-processor-plugin**, conjuntamente con el **maven-compiler-plugin** se añade al **POM.xml** dentro de **<build> <plugins>**, los siguientes elementos **<plugin>**:

```
<!-- *** IMPORTANTE: para no incluir los procesadores como dependencia externa,
se requiere que se compilen en Eclipse y NO se haga un clean para que el
maven-processor-plugin se los encuentre ya compilados en el classpath -->

<!-- ejecuta los procesadores de anotaciones durante la fase generate-sources -->
<!-- NOTA: si se ve la salida de los procesadores sólo en parte o nada en absoluto,
activar la salida de depuración de Maven con su opción -X o, si tenemos definida
una Run Configuration de Eclipse, activar la opción "Debug Output" de la misma -->
<plugin>
    <groupId>org.bsc.maven</groupId>
    <artifactId>maven-processor-plugin</artifactId>
    <version>3.3.2</version>
    <executions>
        <execution>
            <id>process</id>
            <goals>
                <goal>process</goal>
            </goals>
            <phase>generate-sources</phase>
            <configuration>
                <compilerArguments>-source 7 -proc:only -Xlint:-options
                    -XprintRounds -XprintProcessorInfo</compilerArguments>
                <optionMap>
                    <!-- las opciones se pueden poner de la siguiente manera: -->
                    <!-- <nombreOpcion>valorOpcion</nombreOpcion> -->
                    <!-- o -AnombreOpcion=valorOpcion en <compilerArguments> -->
                    <opcion.valor.string>Hola, universo!</opcion.valor.string>
                    <opcion.valor.integer>123456789</opcion.valor.integer>
                </optionMap>
            <processors>
                <!-- no indicamos ningún proc. para que maven-processor-plugin
                    realice el descubrimiento estándar a través del fichero
                    META-INF/services/javax.annotation.processing.Processor -->
                <!-- <processor>nombre.canonico.EjemploProcessor</processor> -->
            </processors>
            <outputDirectory>${basedir}/src/main/java-generated</outputDirectory>
            <outputClassDirectory>${basedir}/target/classes</outputClassDirectory>
        </configuration>
    </execution>
    </executions>
</plugin>
```

```

<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <configuration>
    <source>7</source>
    <target>7</target>
    <!-- habiendo realizado ya el procesamiento,
        lo deshabilitamos en el compilador con -proc:none -->
    <compilerArgument>-proc:none</compilerArgument>
  </configuration>
</plugin>

```

Como se ve en el ejemplo, **maven-processor-plugin** sólo realizará el procesamiento (opción **-proc:only**) pasando las opciones de compilación dadas por **<compilerArguments>**, y pasando a los procesadores las opciones dentro del elemento **<optionMap>**. Como no se han especificado procesadores concretos en el elemento **<processors>**, se realizará el proceso de descubrimiento estándar entre todos los ficheros, directorios y archivos **.jar** que haya en el classpath (formado en el caso de Maven además por todas las dependencias que se resuelvan). Finalmente, se especifica el directorio de salida para los nuevos ficheros fuente generados (**<outputDirectory>**) y el directorio de salida de ficheros de clase (**<outputClassDirectory>**), aunque este último no debería utilizarse, ya que no se va a realizar la compilación final. Para una referencia detallada de todas las demás opciones disponibles, así como otros ejemplos de uso, puede consultarse la documentación del plugin accesible desde su [página web oficial](#).

Para realizar la compilación final de todo el código que se hubiera generado está como segundo **<plugin>** el **maven-compiler-plugin**, que únicamente realizará la compilación final del código generado durante el procesamiento de anotaciones (opción **-proc:none**).

Cuando todo esté correctamente configurado, ejecutando Maven con el goal “**compile**”, podremos ver cómo se ejecuta el procesamiento de anotaciones sobre el código del proyecto, debiendo mostrarse la información de salida de los procesadores por la salida de Maven y, si todo se ha procesado correctamente y sin errores, veremos como el build de Maven termina correctamente con el clásico bloque **BUILD SUCCESS**:

```

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 0.853s
[INFO] Finished at: Tue Jun 03 20:51:31 CEST 2014
[INFO] Final Memory: 6M/98M
[INFO] -----

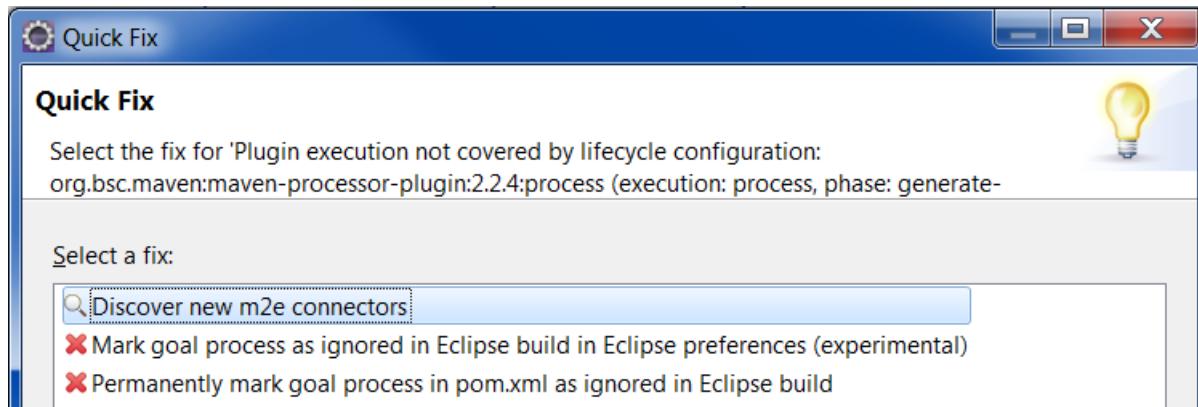
```

maven-processor-plugin: Notas de instalación en Eclipse

Si se usa Eclipse como entorno de desarrollo, al añadir el **maven-processor-plugin** puede ocurrir que obtengamos un error similar al siguiente:

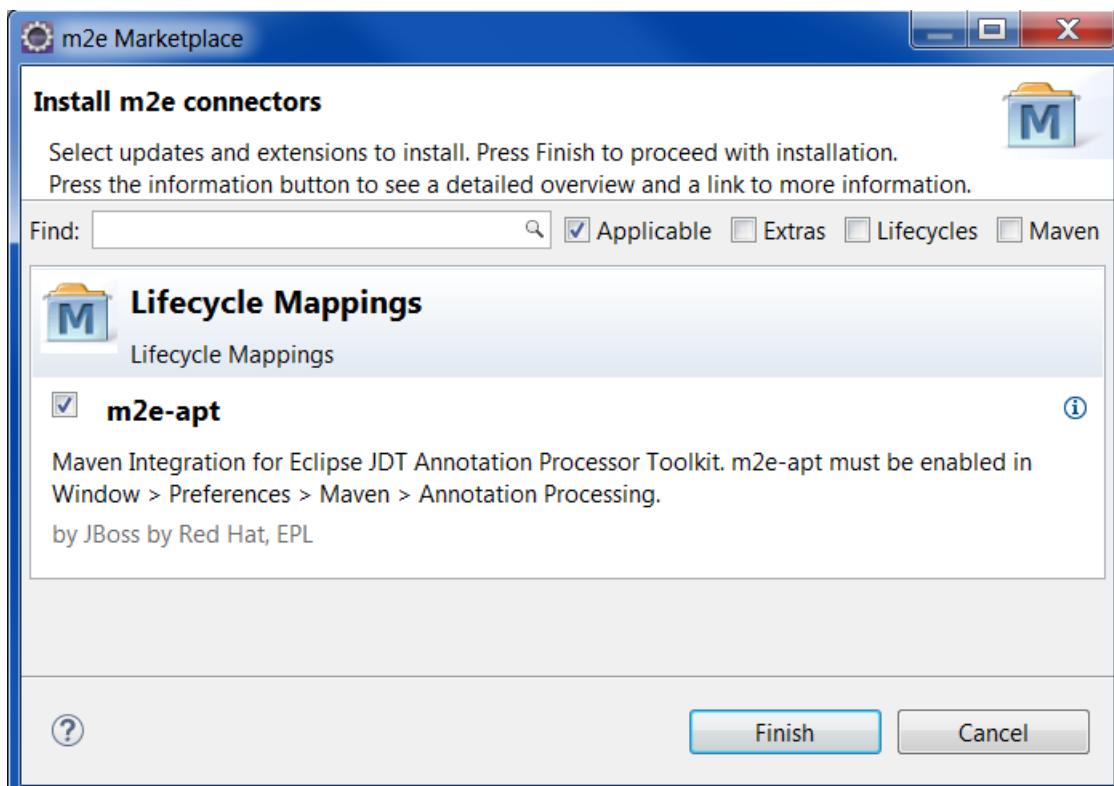
Plugin execution not covered by lifecycle configuration: org.bsc.maven:maven-processor-plugin:3.3.2:process (execution: process, phase: generate-sources)

Esto es así porque Eclipse requiere de la instalación de un conector especial para el plugin **m2e** para la correcta integración de **maven-processor-plugin** dentro de la arquitectura del entorno. Para ello, hacemos click en el error con el botón derecho del ratón y seleccionamos la opción Quick Fix del menú contextual, lo que nos lleva a una lista de varias posibles soluciones para el error. Seleccionamos la primera, “**Discover new m2e connections**”.

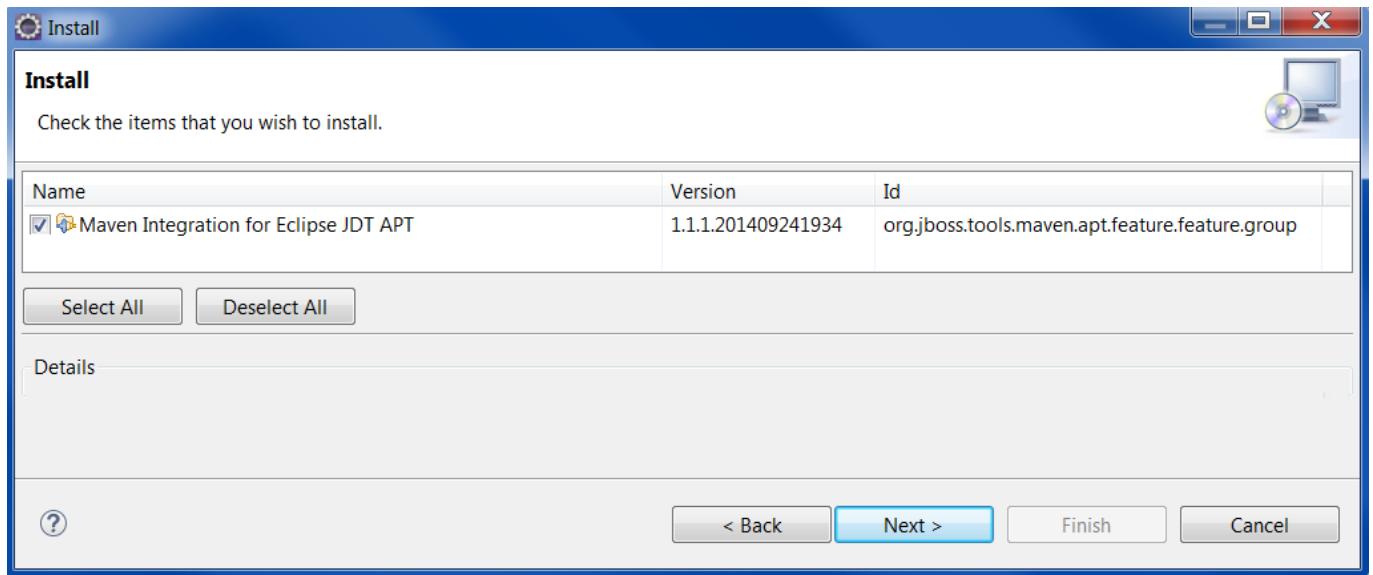


Eclipse buscará conectores para el plugin [m2e](#) de Eclipse. Este plugin es el que permite integrar todos los procesos de Maven dentro de Eclipse y los conectores le permiten extender su funcionalidad para soportar otras funciones avanzadas.

Tras unos momentos, se nos mostrará seleccionable el conector [m2e-apt](#). Este conector permite al plugin m2e soportar procesamiento de anotaciones y se requiere su instalación para poder usar el [maven-processor-plugin](#). Lo seleccionamos y hacemos click en Finish.



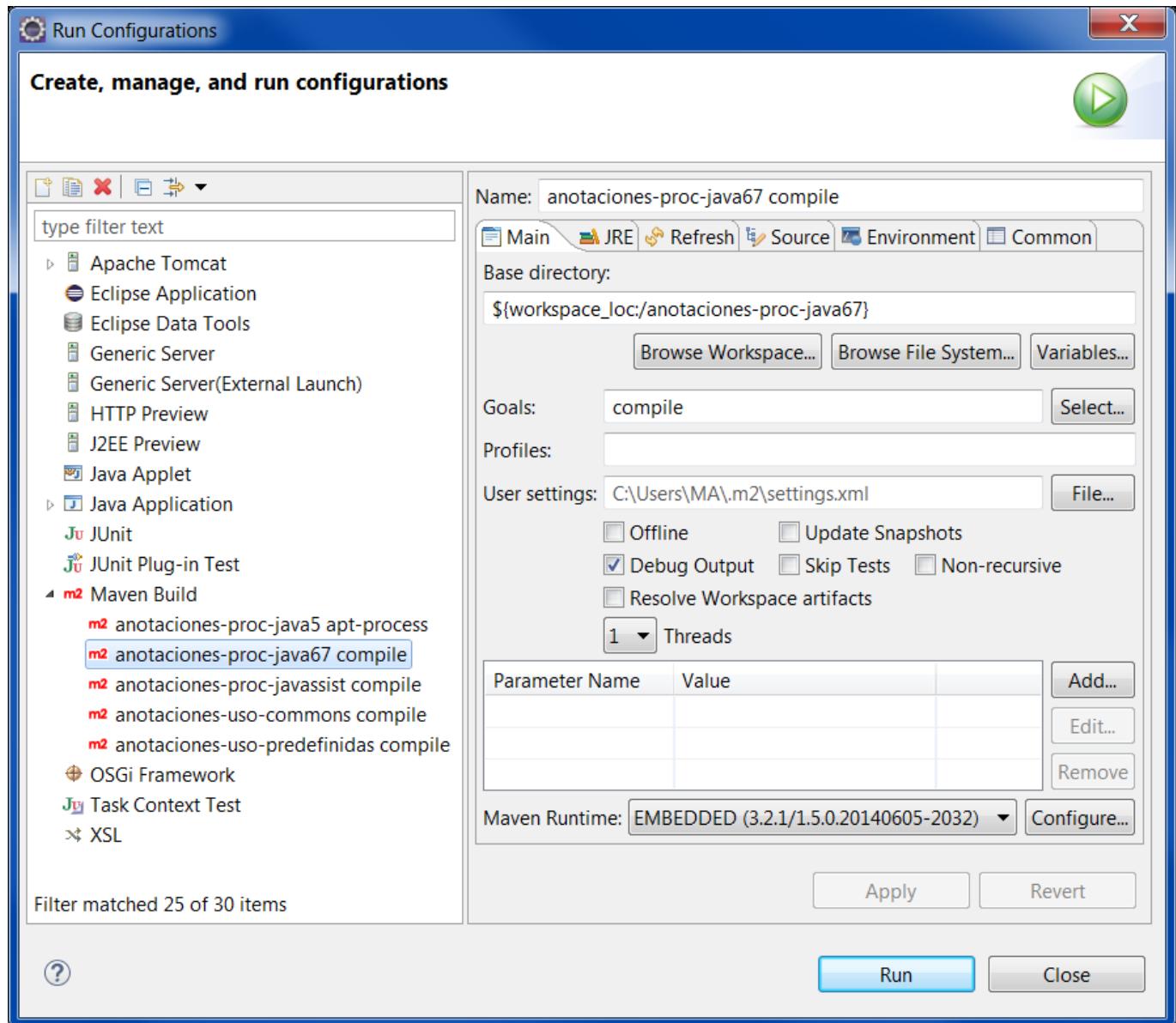
En ese momento, Eclipse nos pedirá confirmación para la instalación de la característica denominada “**Maven Integration for Eclipse JDT APT**”.



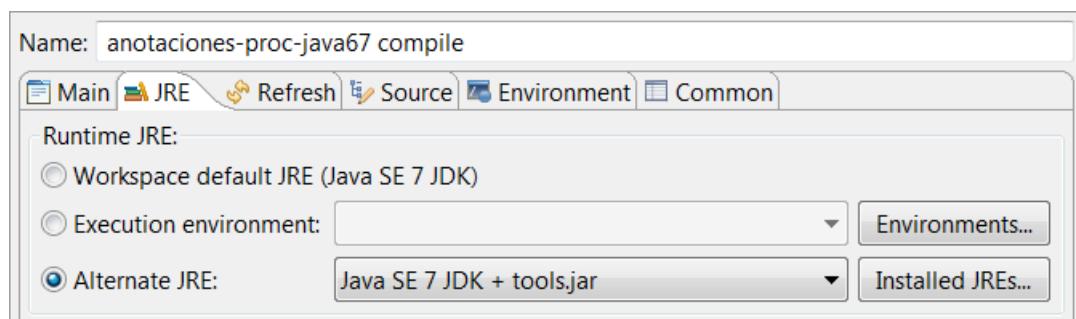
Seguimos el proceso habitual de instalación de características del entorno, y al final nos pedirá reiniciar Eclipse. Tras reiniciar Eclipse podremos comprobar que el error del POM ha desaparecido y ya podremos hacer uso del plugin **maven-processor-plugin** en Eclipse.

maven-processor-plugin: Creando una “Run Configuration” en Eclipse

Para lanzar cómodamente el procesamiento de anotaciones durante el desarrollo y depuración de nuestros procesadores, lo ideal en Eclipse es crear una “Run Configuration”.



En una “Run Configuration” para **maven-processor-plugin** el directorio base debe apuntar al proyecto con el que estamos trabajando y debe estar activada la opción “Debug Output” de Maven. Pueden configurarse más cosas, incluido el paso de parámetros. Finalmente, si estamos usando la **Compiler Tree API** en nuestro procesador, hay que asegurarse de que en la pestaña JRE se apunte a una JDK que hayamos definido y que incluya el fichero “**tools.jar**”.



Procesamiento de anotaciones con `maven-compiler-plugin` (no recomendado)

A diferencia de con `maven-processor-plugin`, con el `maven-compiler-plugin` no es necesariamente una práctica recomendada ni es necesario dividir el proceso en dos, ya que el `maven-compiler-plugin` puede llevar a cabo ambas.

Para usar el `maven-compiler-plugin`, sólo hay que añadir al fichero `POM.xml` del proyecto Java, dentro del elemento XML `<build> <plugins>`, el siguiente elemento `<plugin>`:

```
<!-- *** IMPORTANTE: para no incluir los procesadores como dependencia externa,
se requiere que se compilen en Eclipse y NO se haga un clean para que el
maven-processor-plugin se los encuentre ya compilados en el classpath -->
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <source>7</source>
        <target>7</target>
        <!-- nivel de información en la salida -->
        <showDeprecation>true</showDeprecation>
        <showWarnings>true</showWarnings>
        <verbose>false</verbose>
        <debug>true</debug>
        <!-- opciones al compilador (incluyendo opciones a los procesadores) -->
        <compilerArgs>
            <!-- las opciones se pueden poner de la siguiente manera: -->
            <!-- <arg>opcionLineaComando</arg> -->
            <arg>-Xlint:-options</arg>
            <arg>-XprintRounds</arg>
            <arg>-XprintProcessorInfo</arg>
            <arg>-Aopcion.valor.string=;Hola, universo!</arg>
            <arg>-Aopcion.valor.integer=123456789</arg>
        </compilerArgs>
        <!-- procesadores de anotación -->
        <!-- si comentamos el bloque <annotationProcessors>
        maven-compiler-plugin realizará el proceso de descubrimiento
        estándar a través de los ficheros
        META-INF/services/javax.annotation.processing.Processor -->
        <!--
        <annotationProcessors>
            <processor>nombre.canonico.EjemploProcessor</processor>
        </annotationProcessors>
        -->
        <!-- salida de los ficheros generados -->
        <generatedSourcesDirectory>${basedir}/src/main/java-generated</generatedSourcesDirectory>
        <useIncrementalCompilation>false</useIncrementalCompilation>
    </configuration>
</plugin>
```

Para una referencia detallada de todas las opciones disponibles, así como otros ejemplos de uso, puede consultarse la documentación del plugin accesible desde su [página web oficial](#).

No se recomienda el `maven-compiler-plugin` porque **no muestra la información de salida de los procesadores de anotaciones correctamente**. Por ejemplo, si activamos las opciones `-XprintRounds` y `-XprintProcessorInfo` en lugar de salir la información en su orden habitual, se filtran los mensajes de los procesadores y se vuelcan todos al final. Esto hace que la información de depuración salga mucho antes y le hace perder buena parte de su utilidad. Esto inhabilita al plugin, como poco, para ser usado durante el desarrollo de los procesadores. Este comportamiento es un bug y parece que no va a ser corregido.

15.3.4.3.5.- Invocar `javac` mediante Eclipse.

Eclipse: Un caso especial

[Eclipse](#) se ha convertido en uno de los IDEs Java más extendidos en la actualidad entre la comunidad de desarrollo de la plataforma Java. A la hora de invocar `javac`, Eclipse puede ser el IDE Java que comporte precisamente más problemas debido a que Eclipse implementa su propio compilador Java: el Eclipse Java Compiler o `ecj`, distinto de la implementación de referencia, el clásico `javac` incluido en la JDK de Sun/Oracle. Esto se debe a que los diseñadores de Eclipse no estaban conformes con las limitaciones del `javac` original, que no tenía soporte para algunas características que se querían incorporar en Eclipse.

Funfamental fue que `javac` no soportara [compilación incremental](#), i.e., poder compilar sólo partes del código de una unidad de compilación (clase, módulo, paquete, etcétera), en lugar de la unidad completa. El compilador Java de Eclipse sí soporta compilación incremental con una operación llamada “reconciliación” o “reconciliar” (“reconcile”), lo cual da a Eclipse una mayor eficiencia, así como otra serie de características de las que otros IDEs Java carecen.

Implementación propia del procesamiento de anotaciones en Eclipse

Así pues, en base a lo anterior, ocurre que, cada vez que se implementa alguna nueva característica al lenguaje Java, a su plataforma de desarrollo, o se modifica la Especificación del Lenguaje Java (Java Language Specification o JLS), además de que deba implementarse en la implementación de referencia (Reference Implementation o RI), que es la de Oracle, luego debe implementarse en las demás implementaciones de la Máquina Virtual y el Entorno de Ejecución de la plataforma Java.

Como Eclipse tiene su propio compilador Java, esto hace que, cada vez que se incorpora una novedad al lenguaje o a la plataforma de desarrollo Java que afecta al compilador, la comunidad de desarrollo de Eclipse tenga que ponerse manos a la obra desarrollando una implementación propia que, sólo por ser distinta, aunque trate de emular en todo caso el comportamiento de la implementación de referencia, puede ofrecer resultados diferentes o incluso errores.

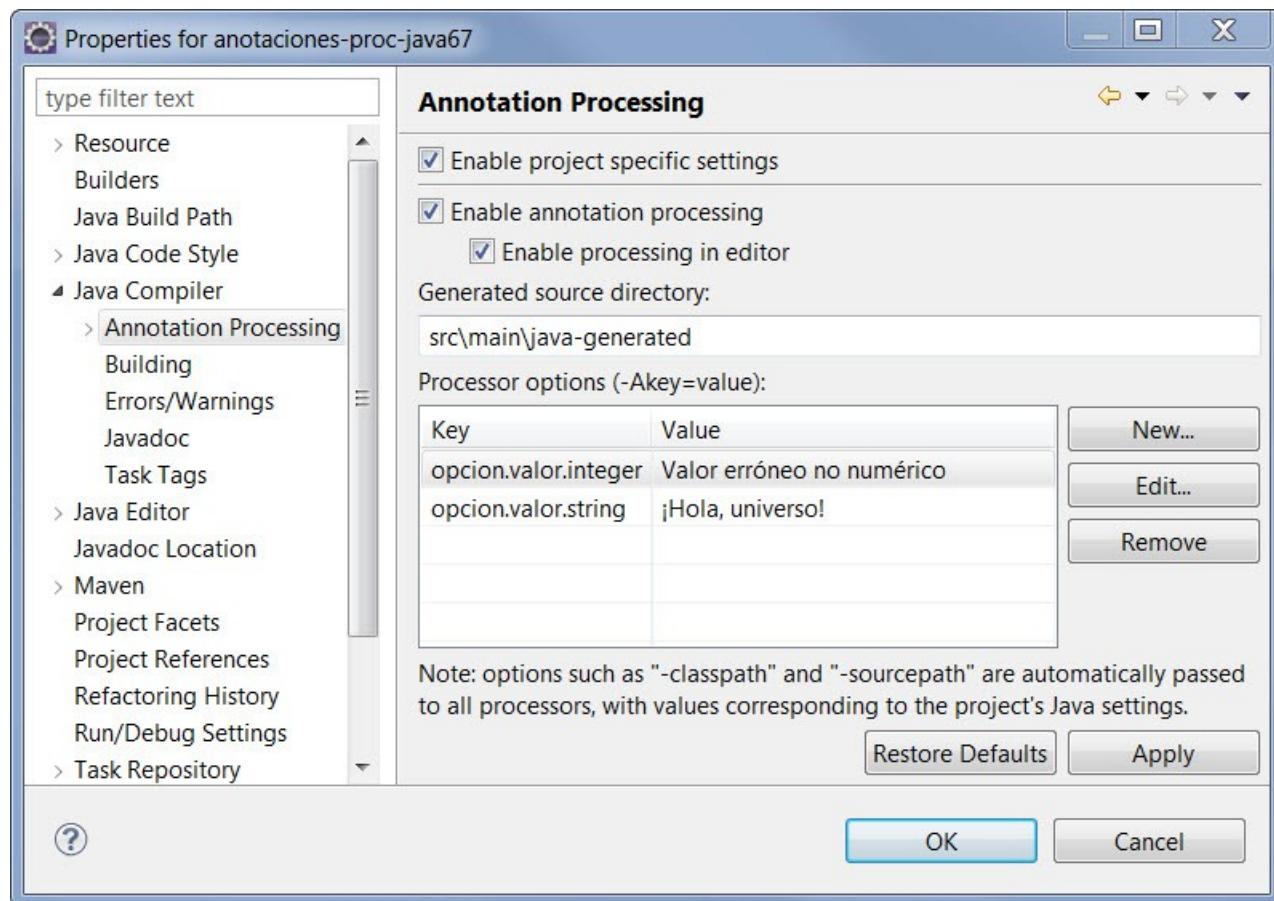
Es por esto que el soporte de procesamiento de anotaciones en Eclipse puede tener un comportamiento distinto al esperado e incluso bugs y errores propios. Debido a que todas las características que se introduzcan en la plataforma Java deben ser implementadas por Eclipse, ocurre que siempre hay un periodo de adaptación desde que sale una nueva versión de la plataforma Java hasta que Eclipse soporta plenamente y con un alto grado de estabilidad todas sus características.

En el caso de Eclipse y el procesamiento de anotaciones JSR 269, no hay buenas noticias, ya que dicho soporte de momento nunca ha estado libre de limitaciones y errores importantes que, a día de hoy, no han sido aún corregidos de raíz.

Por tanto, aunque en el presente manual se incluye este apartado por razones didácticas y de completitud, [**no se recomienda el uso de las facilidades de Eclipse para soporte de anotaciones JSR 269. Se recomienda usar Maven integrado en Eclipse**](#), para lo cual puede consultarse el subapartado anterior “Invocación de `javac` mediante Maven”.

Eclipse: Configuración del procesamiento de anotaciones JSR 269

El soporte de procesamiento de anotaciones JSR 269 en Eclipse se configura accediendo a las propiedades del proyecto, apartado “**Java Compiler**” / “**Annotation Processing**”:



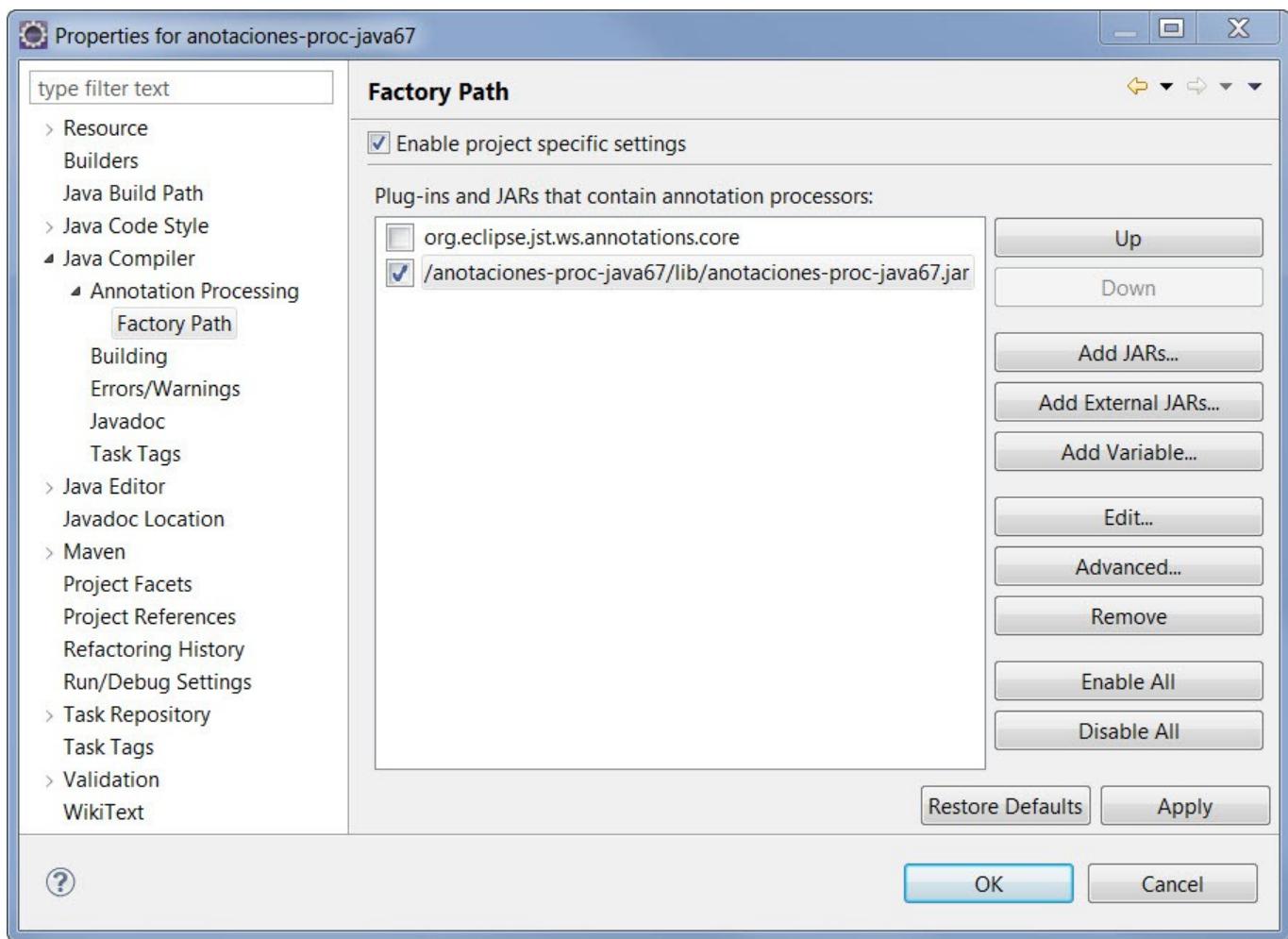
Marcando “**Enable project specific settings**” y “**Enable annotation processing**” activaremos el procesamiento de anotaciones para nuestro proyecto, pero no basta sólo con eso, claro. Hay algunas cosas más todavía que será necesario configurar.

En “**Generated source directory**”, podemos ajustar el directorio de salida para los nuevos ficheros de código fuente (opción `-s DIR_FUENTES_GENERADAS` de `javac`). El valor por defecto es “`.apt_generated`”, no el que hemos configurado nosotros (**CUIDADO**: al empezar por “.” podría ser que el directorio se oculte en algunas vistas del entorno). También disponemos de la posibilidad de pasar una lista de opciones a los procesadores introduciendo la clave y el valor directamente. Nótese que el valor establecido para la opción “`opcion.valor.integer`” será un valor erróneo para el procesador, ya que no es un valor numérico.

Otra cosa importante es que en la integración del procesamiento JSR 269 en Eclipse, como se indica en la nota de la parte de abajo del cuadro de diálogo, las opciones de `-classpath` (y el menos utilizado `-sourcepath`, no confundir con `-s`) no podemos establecerlas nosotros, si no que se pasan automáticamente en función de la configuración del proyecto. Esto no debería ser un problema, ya que Eclipse permite configurar el `CLASSPATH` de un proyecto con bastante detalle en el apartado “Java Build Path” de sus propiedades.

Eclipse no descubre procesadores en CLASSPATH, sólo en FACTORYPATH

El proceso de descubrimiento de procesadores de anotaciones es quizás lo que peor está resuelto en Eclipse debido a que sólo se busca en los archivos `.jar` configurados en el apartado “Java Compiler” / “Annotation Processing” / “Factory Path”:



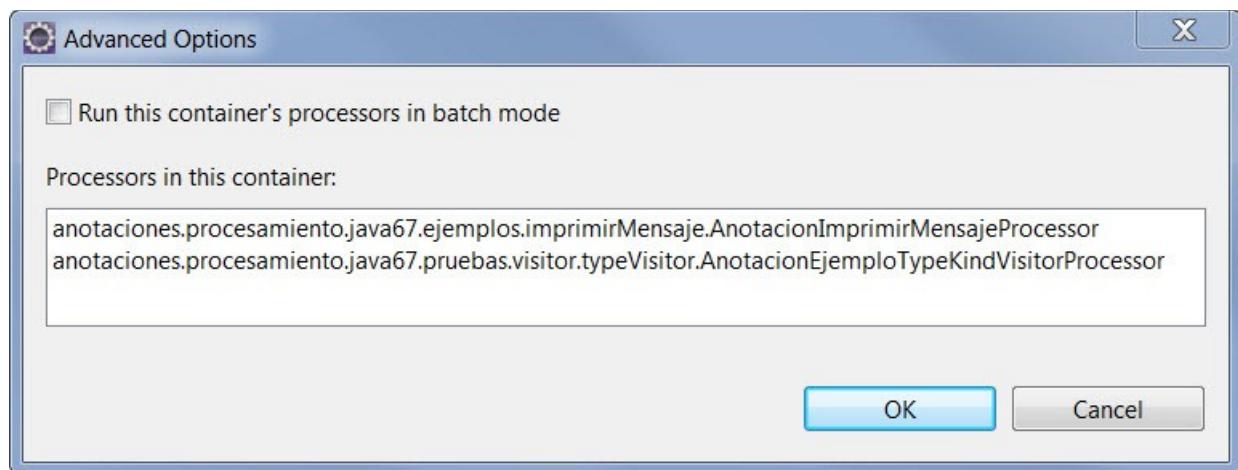
Esta falta de flexibilidad no es tal cuando invocamos a `javac` por línea de comandos o vía Maven, ya que en todos los casos excepto en Eclipse se lanza el proceso de descubrimiento de procesadores que por defecto busca los procesadores en el `CLASSPATH` indicado (en el caso de `javac`) o en el del proyecto (en el caso de Maven). No obstante, como hemos comentado, el compilador de Eclipse implementa su propio proceso de descubrimiento diferente al de `javac`, y en el que únicamente busca procesadores en su propio `FACTORYPATH`, que serían los `.jar` que se añaden en este apartado “Factory Path”, y no busca en el `CLASSPATH` como tal del proyecto.

Que el proceso de descubrimiento de procesadores de Eclipse no busque en el `CLASSPATH` del proyecto, sino sólo en el `FACTORYPATH`, puede en principio parecer algo insignificante, pero tiene un inconveniente muy importante a nivel práctico de cara a trabajar en el desarrollo de procesadores de anotaciones: **es necesario exportar el propio proyecto a un .jar para que Eclipse descubra las factorías de nuestros propios procesadores. Pero lo peor de todo no es eso, si no que, si se modifica cualquier procesador, hay que volver a exportar el .jar ante cualquier modificación que se haga, por pequeña que sea.** Por este motivo, en el “Factory Path” de la imagen anterior aparece incluido el fichero `.jar` exportado del propio proyecto en cuestión (`anotaciones-proc-java67.jar`).

El tener que estar exportando el proyecto a `.jar` continuamente cada vez que se quiere probar cualquier mínima modificación sobre un procesador de anotaciones, como es lógico, es todo un engorro y, por ello, por ejemplo, **en el día a día del desarrollo de los ejemplos de procesamiento de anotaciones vinculados a este manual se ha venido trabajando con Maven**, que a este respecto resulta mucho más práctico al estar integrado también en Eclipse, pero, al invocar a `javac`, usa el mecanismo de descubrimiento de procesadores de `javac` que sí busca en el **CLASSPATH** de Maven, que es el **CLASSPATH** del proyecto de Eclipse. Este problema sólo es realmente grave si somos desarrolladores de procesadores, pero no si somos simplemente usuarios de procesadores de terceros, ya desarrollados y probados, y nosotros únicamente tengamos que añadir su `.jar` para usarlos.

Comprobar el resultado del descubrimiento de procesadores en los `.jar` añadidos

Cada vez que incluyamos un `.jar` en el “Factory Path” de Eclipse, es recomendable seleccionarlo y hacer click en el botón “**Advanced...**”. Esto nos mostrará un cuadro de diálogo “**Advanced Options**” con los procesadores encontrados dentro de dicho fichero `.jar`.



Es una práctica recomendada hacer click en “Advanced...” y comprobar que, cuando añadimos un fichero `.jar` al “Factory Path”, Eclipse haya sido capaz de descubrir en dicho fichero todas las factorías de procesadores esperadas.

Si no apareciera ninguna factoría, probablemente se deba a que el archivo `.jar` no está correctamente generado, o a que carezca del directorio de recursos /META-INF/services con el fichero que lista las clases que implementan los procesadores de anotaciones JSR 269, que recordemos que debe ser el siguiente:

/META-INF/services/javax.annotation.processing.Processor

CUIDADO: No todos los `.jar` añadidos a la “Factory Path” han de contener factorías de procesadores de anotaciones necesariamente. La **FACTORYPATH** de Eclipse también tendrá que incluir los `.jar` de cualquiera otras librerías que los procesadores necesiten para reconocer las clases que procesan, así como para poder procesarlas. Por ejemplo: si añadiéramos un procesador de anotaciones que trabaje sobre las anotaciones de JPA ([Java Persistence API](#)), una API de persistencia que hace un uso muy intensivo de anotaciones, deberemos añadir al **FACTORYPATH** también el `.jar` con las clases de JPA propiamente dicha, o el procesamiento arrojará excepciones **ClassNotFoundException** para cada una de las clases de JPA, ya que no serán reconocidas.

CUIDADO: En el ámbito de la ejecución del procesamiento de anotaciones, tal y como está implementado en Eclipse, es como si el **CLASSPATH** no existiera, sólo existe el **FACTORYPATH**. Es decir, que **Eclipse no es capaz de descubrir procesadores de anotaciones en su CLASSPATH, como hace javac en su proceso de descubrimiento estándar**. Este es un bug de Eclipse [Bug 280542] desde hace mucho tiempo que parece ser que aún no ha sido posible corregir y que supone severas limitaciones e incomodidades a la hora de trabajar con el procesamiento de anotaciones implementado en dicho entorno. Es por este motivo que se usa Maven como método preferido para la ejecución del procesamiento.

En “**Advanced Options**” aparece la opción “**Run this container's processors in batch mode**” (“Ejecutar los procesadores de este contenedor en modo por lotes”). Cuando está activada, todo el procesamiento funcionará de manera que mejore la compatibilidad con **apt**, la herramienta de procesamiento de anotaciones J2SE 1.5. Para más información sobre el soporte de **apt** en Eclipse, puede consultarse el subapartado “Invocación de **apt** mediante Eclipse” del apartado “Procesamiento J2SE 1.5”. La opción de ejecución de procesadores en modo por lotes sólo afecta a los procesadores J2SE 1.5 que usan **apt**, y **no afecta a los procesadores de anotaciones JSR 269**, por lo que no incidiremos sobre ella y, en caso de estar usando procesadores JSR 269 nos limitaremos a ignorarla.

Obtención de resultados del proc. JSR 269 en Eclipse: Malas noticias

Finalmente, cuando hayamos completado toda la configuración de Eclipse correctamente, podremos ver que, si los procesadores se ejecutan y empiezan a emitir warnings o errores, estos podrían aparecer en la vista “**Problems**”, en la vista “**Markers**” o en la vista “**Error Log**”. **IMPORTANTE:** Este es un problema grave que tiene ahora mismo el entorno de desarrollo Eclipse. Se ha probado en varias de las últimas versiones del entorno (Kepler y Luna) y se ha exhibido en ambas el mismo comportamiento, diferente del esperado y del que ocurre en el caso del procesamiento J2SE 1.5. Para el caso del procesamiento de anotaciones JSR 269, tanto en Eclipse Kepler como en Eclipse Luna, “**Problems**” aparece vacío. “**Markers**” sólo muestra los warnings bajo la categoría “APT Problems”, y donde sí aparecen todos los mensajes es en “**Error Log**”, pero todos con un nivel de severidad “Info”, incluso el error que hemos provocado pasando un valor incorrecto para la opción “**opcion.valor.integer**”.

El siguiente es el contenido referido mostrado por las vistas “**Problems**” (totalmente vacía) y “**Markers**” (sólo con algunos mensajes de severidad “Warning” como “APT Problems”):

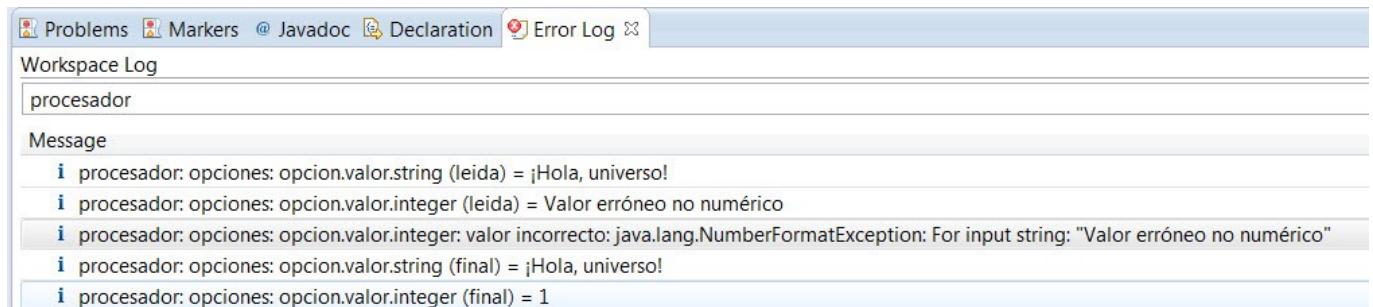


Problems		
0 items		
Description	Resource	Path

Markers		
0 errors, 11 warnings, 1 other		
Description	Resource	Path
APT Problems (2 items)		
PROCESAMIENTO DE LA CLASE: anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.ClaseAnotadaAnotacionImprimirMensaje.java	ClaseAnotadaAnotacionImprimirMensaje.java	/anotaciones-proc-java67/src/main/java/anotaciones/procesamiento/java67/ejemplos/imprimirMensaje/ClaseAnotadaAnotacionImprimirMensaje.java
Valor: Definicion de la clase.	ClaseAnotadaAnotacionImprimirMensaje.java	/anotaciones-proc-java67/src/main/java/anotaciones/procesamiento/java67/ejemplos/imprimirMensaje/ClaseAnotadaAnotacionImprimirMensaje.java

El hecho de que **Eclipse no procesa correctamente la salida de los procesadores de anotaciones JSR 269** es un error, puesto que la compilación, si se emite algún error, debería fallar directamente. Hay un bug abierto acerca de este problema [[Bug 428357](#)] que está pendiente de ser corregido. Desgraciadamente, además de ese, el componente de procesamiento de anotaciones en Eclipse, llamado [JDT APT](#) tiene una buena cantidad de [bugs pendientes](#).

En la siguiente imagen puede apreciarse el contenido de la vista “**Error Log**” filtrado para que se muestren los mensajes cuyo texto contenga la palabra “procesador”. Nótese como se muestra el mensaje de error provocado al arrojar el procesador una **NumberFormatException** debido a que se ha pasado un valor no numérico para la opción “**opcion.valor.integer**” y, debido a ello, como dicha opción toma el valor final por defecto de 1.



The screenshot shows the Eclipse IDE interface with the "Error Log" tab selected in the top navigation bar. Below the tabs, there is a search bar with the text "procesador". Under the search bar, there is a section titled "Message" containing several log entries:

- i procesador: opciones: opcion.valor.string (leida) = ¡Hola, universo!
- i procesador: opciones: opcion.valor.integer (leida) = Valor erróneo no numérico
- i procesador: opciones: opcion.valor.integer: valor incorrecto: java.lang.NumberFormatException: For input string: "Valor erróneo no numérico"
- i procesador: opciones: opcion.valor.string (final) = ¡Hola, universo!
- i procesador: opciones: opcion.valor.integer (final) = 1

Así pues, el error de compilación emitido por el procesador sólo se muestra únicamente en la vista “**Error Log**”, una vista poco utilizada por los desarrolladores en general y, para colmo, no se trata como un error de compilación, lo que hace que dicha compilación no falle, cuando obviamente debería hacerlo.

IMPORTANTE: Debido a las limitaciones y errores exhibidos por el actual soporte del procesamiento JSR 269 en Eclipse, desgraciadamente hemos de desaconsejar la utilización del soporte interno de Eclipse para la realización de procesamiento de anotaciones JSR 269.

En lugar de utilizar el soporte interno de Eclipse para el procesamiento de anotaciones JSR 269, **recomendamos el uso de Maven que, como hemos visto en el subapartado anterior, también puede integrarse en Eclipse**.

NOTA: Los ejemplos de código de este manual vienen suministrados en proyectos de Eclipse. Para más información sobre cómo invocar el procesamiento de anotaciones JSR 269 con Maven, puede consultarse el subapartado anterior “Invocación de `javac` mediante Maven”.

15.4.- Procesamiento JSR 269: Paso a paso.

Hasta ahora, hemos visto una visión global del procesamiento de anotaciones JSR 269, así como una inmersión detallada en cada uno de sus componentes para tener una referencia completa de los mismos antes de empezar a desarrollar nuestros procesadores de anotaciones.

Pues bien, ya estamos en disposición de ponernos manos a la obra. Así pues, en este apartado describiremos paso a paso, desde el principio hasta el final, el procedimiento habitual que se sigue para implementar el procesamiento de anotaciones JSR 269.

15.4.1.- Análisis y Diseño del tipo anotación.

El requisito de implementar un procesamiento de anotaciones siempre surje de alguna necesidad concreta, que es analizada y cuyo análisis llega a la conclusión de que la solución óptima para cumplir dicha necesidad sería cubierta a través del procesamiento de anotaciones.

El primer paso es, por tanto, como en cualquier nuevo desarrollo software, llevar a cabo un análisis de los requisitos que ha de cumplir el nuevo tipo anotación y realizar un diseño de dicho tipo anotación de forma que se ajuste a dichos requisitos.

En función de las requisitos planteados, deberán decidirse todos los detalles pertinentes, como, por ejemplo, sobre qué elementos de código se aplicará el tipo anotación (`@Target`), cuál debería ser su política de retención (`@Retention`), si no tendrá elementos (anotación marcadora) o, si necesita elementos, qué elementos concretamente y de qué tipos serían los más apropiados.

El diseño de un tipo anotación podrá ser más o menos sencillo dependiendo del número y la complejidad de los requisitos a satisfacer. Sin embargo, en general, un principio de diseño que habría que tratar de seguir en lo posible es que el tipo anotación debería permanecer sencillo para alentar su uso por parte de los usuarios a los que vaya dirigido. Está claro que las anotaciones marcadoras (sin elementos) o las anotaciones sencillas (con un único elemento) son las más atractivas por la comodidad de su uso. No obstante, si los requisitos lo imponen, podría necesitarse definir un gran número de elementos, como ocurre en el caso del tipo anotación `@DataSourceDefinition`, con casi una veintena de elementos.

La principal prioridad a la hora de diseñar un tipo anotación será tratar de ajustarse siempre lo mejor posible a los requisitos. En nuestro caso, para mantener este primer ejemplo sencillo, vamos a diseñar un tipo anotación que podrá aplicarse a cualquier tipo de declaración y que simplemente servirá para imprimir un mensaje, así que únicamente incluirá un elemento de tipo `String` para el mensaje con un valor por defecto:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface ImprimirMensaje {
    String value() default ";Hola, mundo! (mensaje por defecto)";
}
```

15.4.2.- Implementación del Procesador de anotaciones.

Una vez concluido el análisis de requisitos y el diseño del tipo anotación a procesar, se puede empezar con la implementación del procesador de anotaciones correspondiente. Para ello, como hemos visto, dispondremos de las facilidades de la Language Model API y la API de procesamiento JSR 269, así como del soporte de procesamiento dado por `javac`.

El primer paso y más importante en la implementación de un procesador de anotaciones es decidir la estrategia a seguir para descubrir los elementos a procesar. En el caso de nuestro tipo anotación `@ImprimirMensaje`, no hemos establecido ningún requisito concreto sobre el orden en el que pueden procesarse e imprimirse los mensajes, así que optaremos por utilizar `getElementsAnnotatedWith` para recuperar los elementos anotados a procesar directamente.

El código fuente completo y profusamente comentado del procesador de anotaciones para el tipo anotación `@ImprimirMensaje`, implementado en la clase `ImprimirMensajeProcessor` puede encontrarse en el material complementario que se acompaña junto con este manual, dentro del proyecto `anotaciones-proc-java67`.

No obstante, aunque no es posible incluir el código fuente completo del procesador por cuestiones de espacio, sí vamos a comentar algunas cuestiones de especial interés.

En este primer procesador utilizaremos la estrategia más sencilla: obtener directamente los elementos anotados a procesar a través de `getElementsAnnotatedWith`, iteramos sobre ellas en un bucle `for` y las vamos procesando en un método auxiliar `procesarElemento` según sea su tipo en la rama de un `if` de gran tamaño. Así pues, el método `process` quedaría así:

```
@Override
public boolean process(Set<? extends TypeElement> tiposAnotacion, RoundEnvironment roundEnv) {
    // PROCESAMIENTO SEGÚN LA RONDA EN LA QUE NOS ENCONTREMOS

    if (roundEnv.processingOver()) {
        // PROCESAMIENTO DE LA RONDA FINAL

        // simplemente presentamos un mensaje informativo para
        // mostrar que el procesador ha finalizado su procesamiento
        this.message.printMessage(Kind.NOTE, "procesador: "+
            this.getClass().getCanonicalName() + " finalizado.");
    } else {
        // PROCESAMIENTO DE LOS ELEMENTOS ANOTADOS

        // iteramos sobre la lista de elementos anotados a procesar
        for (Element elemento : roundEnv.getElementsAnnotatedWith(this.claseAnotacion)) {
            // PROCESAMIENTO DEL ELEMENTO ANOTADO

            // procesamos el elemento anotado
            procesarElemento(elemento);
        }
    }

    // finalizamos retornando false porque no queremos quedarnos con el procesado
    // de este tipo anotación en exclusiva, si devolvemos true el conjunto de
    // anotaciones que hemos procesado no será pasado a otros procesadores
    return false;
}
```

Y el método `procesarElemento` sería una rama `if` de gran tamaño donde procesaremos cada elemento en función de su tipo concreto, dado por el método `getKind`. Esto es así porque no hemos querido introducir en este primer ejemplo el uso de los Visitor, si no que hemos querido ilustrar la forma más sencilla de construir la lógica de descubrimiento y procesamiento de los elementos anotados.

```

private void procesarElemento(Element elemento) {
    // REFERENCIA A LA ANOTACIÓN DEL ELEMENTO ANOTADO

    // como sabemos que el elemento está anotado con el tipo anotación a procesar,
    // debido a que ha sido retornado por el método getElementsAnnotatedWith,
    // recuperamos la anotación que queremos procesar
    ImprimirMensaje anotacion =
        elemento.getAnnotation(this.claseAnotacion);

    // si el elemento no estuviera anotado (por ser una llamada recursiva)
    // -> ignoramos el elemento retornando directamente
    if (anotacion == null) return;

    // PROCESAMIENTO DEL ELEMENTO ANOTADO SEGÚN SU TIPO

    // NOTA: A diferencia del procesamiento en J2SE 1.5 con la Mirror API,
    // donde había que tener en cuenta la jerarquía de las declaraciones
    // a la hora de montar el macro-if, en el procesamiento JSR 269,
    // al interrogar por el ElementKind, que es un tipo enumerado muy
    // específico, podemos preguntar directamente por el tipo del elemento
    // en cualquier orden, sin necesidad de preocuparnos por la jerarquía
    // de herencia y las relaciones entre los diferentes tipos.

    if (elemento.getKind() == ElementKind.PACKAGE) {
        // PAQUETE ANOTADO

        // down-cast
        PackageElement elementoPq = (PackageElement) elemento;

        // imprimimos su información
        this.messager.printMessage(Kind.NOTE,
            "PAQUETE ANOTADO", elementoPq);
        this.messager.printMessage(Kind.NOTE,
            "PAQUETE: " + elementoPq.getQualifiedName());
        this.messager.printMessage(Kind.NOTE,
            "MENSAJE: " + anotacion.value());

    } else if (elemento.getKind() == ElementKind.CLASS) {
        // CLASE ANOTADA

        // down-cast
        TypeElement elementoClase = (TypeElement) elemento;

        // imprimimos su información
        this.messager.printMessage(Kind.NOTE,
            "CLASE ANOTADA", elementoClase);
        this.messager.printMessage(Kind.NOTE,
            "CLASE: " + elementoClase.getQualifiedName());
        this.messager.printMessage(Kind.NOTE,
            "MENSAJE: " + anotacion.value());

        // PROCESAMIENTO DE LOS INICIALIZADORES DE LA CLASE

        // procesamos las variables dentro de los inicializadores
        procesarInicializadores(elementoClase);
    }
}

```

```

} else if (elemento.getKind() == ElementKind.ENUM) {
    // ... PROCESAMIENTO DE ENUMERADO ANOTADO ...

} else if (elemento.getKind() == ElementKindINTERFACE) {
    // ... PROCESAMIENTO DE INTERFAZ ANOTADA ...

} else if (elemento.getKind() == ElementKindANNOTATION_TYPE) {
    // ... PROCESAMIENTO DE TIPO ANOTACIÓN ANOTADO ...

} else if (elemento.getKind() == ElementKindFIELD) {
    // ... PROCESAMIENTO DE CAMPO ANOTADO ...

} else if (elemento.getKind() == ElementKindENUM_CONSTANT) {
    // ... PROCESAMIENTO DE CONSTANTE ENUMERADO ANOTADO ...

} else if (elemento.getKind() == ElementKindPARAMETER) {
    // ... PROCESAMIENTO DE PARÁMETRO ANOTADO ...

} else if (elemento.getKind() == ElementKindLOCAL_VARIABLE) {
    // ... PROCESAMIENTO DE VARIABLE LOCAL ANOTADA ...

} else if (elemento.getKind() == ElementKindEXCEPTION_PARAMETER) {
    // ... PROCESAMIENTO DE PARÁMETRO DE EXCEPCIÓN ANOTADO ...

} else if (elemento.getKind() == ElementKindRESOURCE_VARIABLE) {
    // VARIABLE DE RECURSO ANOTADA (*** NUEVO ELEMENTO DE JAVA SE 7 ***)
    // ... PROCESAMIENTO DE VARIABLE DE RECURSO ANOTADA ...

} else if (elemento.getKind() == ElementKindMETHOD) {
    // MÉTODO ANOTADO

    // down-cast
    ExecutableElement elementoMetodo = (ExecutableElement) elemento;

    // imprimimos su información
    this.messager.printMessage(Kind.NOTE,
        "MÉTODO ANOTADO", elementoMetodo);
    this.messager.printMessage(Kind.NOTE,
        "MÉTODO: " + elementoMetodo.getSimpleName());
    this.messager.printMessage(Kind.NOTE,
        "MENSAJE: " + anotacion.value());

    // VARIABLES DEL MÉTODO

    // procesamos las variables del método
    procesarVariablesElementoEjecutable(elementoMetodo);

} else if (elemento.getKind() == ElementKindCONSTRUCTOR) {
    // ... PROCESAMIENTO DE CONSTRUCTOR ANOTADO ...

```

```

} else if (elemento.getKind() == ElementKind.STATIC_INIT) {

    // INICIALIZADOR ESTÁTICO ANOTADO

    // NO ANOTABLE, NO PARSEABLE A Element
    // DESCUBRIBLE SÓLO VÍA Compiler Tree API

} else if (elemento.getKind() == ElementKind.INSTANCE_INIT) {

    // NO ANOTABLE, NO PARSEABLE A Element
    // DESCUBRIBLE SÓLO VÍA Compiler Tree API

} else if (elemento.getKind() == ElementKind.TYPE_PARAMETER) {

    // NO ANOTABLE
}

} // procesarElemento

```

NOTA: Debido a su gran longitud y monotonía, en el código anterior se han suprimido los bloques de código similares o carentes de interés (señalados con puntos suspensivos `// ...`). Para ver el código completo, acúdase al material de ejemplo incluido con el presente manual.

Para evitar tener que construir bloques `if` tan grandes, así como el down-cast implícito que conlleva cada una de sus ramas, podemos utilizar algún Visitor. Implementar la lógica de procesamiento en los métodos de los Visitor en lugar de en las ramas de los bloques `if` supone también una importante mejora estructural del código.

Una vez elegida la estrategia de descubrimiento de los elementos anotados a procesar y montada la estructura que nos permita procesar cada tipo de elemento de forma específica según nuestras necesidades, sólo queda escribir el tratamiento adecuado a nuestros requisitos.

En este caso, por ser nuestro primer procesador de anotaciones y por tener el tipo anotación `@ImprimirMensajes` unos requisitos tan sencillos como simplemente imprimir los mensajes por la salida del compilador sin ningún orden específico, el procesamiento de todos los tipos de elementos son muy sencillos y muy parecidos: simplemente mostramos por la salida del compilador una serie de mensajes informativos sobre la posición en el código fuente de los elementos procesados, su nombre y su mensaje asociado. En todos los tipos de elementos se realiza un tratamiento similar, imprimir la información y el mensaje, con algunas salvedades:

- **Clases y tipos enumerados:** Además, se buscan las variables anotadas dentro del cuerpo de sus bloques inicializadores llamando al método `procesarInicializadores`.

- **Métodos y constructores:** Además, se buscan las variables anotadas dentro de sus cuerpos llamando al método `procesarVariablesElementoEjecutable`.

- **Inicializadores estáticos, de instancia y parámetros de tipo:** No se hace nada en dichos bloques, puesto que estos tipos de elementos no son anotables y, por tanto, nunca se podrán descubrir elementos anotados de dichos tipos. El compilador Java dará un error si intentamos anotarlos. Sólo los parámetros de tipos serán anotables, pero a partir de Java SE 8. Hasta Java SE 7, el compilador dará un error si intentamos anotar parámetros de tipo.

15.4.3.- Anotación de elementos de código.

Una vez listo el diseño del tipo anotación y la implementación de su correspondiente procesador, ya es posible la utilización de tipo anotación en el código fuente por parte de los usuarios potenciales a los que vaya destinado.

Como ejemplo, se han creado varios ficheros de código fuente anotado para demostrar todas las formas posibles de utilizar una anotación. El paquete donde se han ubicado el tipo anotación y su procesador, tiene un subpaquete **elementosAnotados** que contiene la definición de un paquete (**package-info.java**), una clase, una interfaz, así como un tipo anotación, para mostrar por la salida del procesador para elementos de todo tipo.

El código de clase anotada definida como ejemplo, **ImprimirMensajeClaseAnotada**, como puede observarse, está definido para crear al menos una declaración de todas las que es posible definir dentro de una clase. Los tipos de declaraciones restantes se han realizado entre los demás ficheros.

Así pues, nótese como en el código de la clase anotada de ejemplo se incluyen anotaciones sobre elementos de código que no se suelen utilizar muy frecuentemente como, por ejemplo: anotaciones sobre un tipo enumerado y sus constantes, sobre parámetros de métodos, y variables locales a los cuerpos de los métodos y los bloques inicializadores, tanto estáticos como de instancia.

Haremos especial hincapié en las anotaciones interiores a los cuerpos de los métodos, que son descubiertas excepcionalmente gracias a las facilidades dadas por la Compiler Tree API. En la clase anotada de ejemplo **ImprimirMensajeClaseAnotada** se define el siguiente método que contiene todos los tipos posibles de anotaciones ubicables dentro de un método. Las anotaciones que sólo son descubribles para su procesamiento a través de la Compiler Tree API son las anotaciones sobre variables locales, sobre parámetros de manejadores de excepciones, así como las anotaciones sobre variables de recurso dentro de las nuevas expresiones [try-with-resources](#) introducidas en Java SE 7 (todas estas últimas aparecen resaltadas en negrita):

```
@ImprimirMensaje("Método impimirMensajeAFichero.")
public void impimirMensajeAFichero(
    @ImprimirMensaje("Parámetro mensaje del método impimirMensajeAFichero.") String mensaje,
    @ImprimirMensaje("Parámetro rutaFichero del método impimirMensajeAFichero.") String rutaFichero)
{
    @ImprimirMensaje("Variable local.")
    String variableLocal = "Mensaje: ";

    try (@ImprimirMensaje("Variable de recurso.") BufferedWriter escritorFichero =
        new BufferedWriter(new FileWriter(new File(rutaFichero)))) {
        // escribimos al fichero
        escritorFichero.write(variableLocal + mensaje);

    } catch (@ImprimirMensaje("Parámetro de manejador de excepción.") IOException e) {
        // imprimimos el error
        System.out.println("impimirMensaje: error: " + e);
    }
}
```

15.4.4.- Invocación del procesamiento y obtención de resultados.

Una vez diseñado el tipo anotación, implementado su correspondiente procesador y anotado código cliente para ser procesado, ya sólo falta la última etapa: la invocación efectiva del procesamiento de anotaciones y la obtención de sus resultados.

Aunque es posible invocar a **javac** para realizar el procesamiento de múltiples formas, como hemos visto en el apartado “Invocación de **javac**”, utilizaremos el más clásico: la línea de comandos, para así ilustrar su uso siguiendo más de cerca la documentación oficial.

En nuestro caso, para invocar nuestro procesamiento de anotaciones, ejecutaremos desde el directorio raíz del proyecto Eclipse llamado **anotaciones-proc-java67** este comando:

```
javac -version -source 7 -proc:only -Xlint:-options ^
-XprintProcessorInfo -XprintRounds ^
-cp .\target\classes ^
-s .\src\main\java-generated ^
-processor anotaciones.procesamiento.java67.ejemplos.ImprimirMensaje.ImprimirMensajeProcessor ^
.\src\main\java\anotaciones\procesamiento\java67\ejemplos\imprimirMensaje\elementosAnotados\*.java
```

Siendo el primer ejemplo de procesamiento, se han incluido en el comando de invocación a **javac** todas las opciones no estándar de depuración (las que empiezan por **-X**). De esta forma, el lector podrá constatar y realizar un seguimiento por sí mismo del modo de operación que sigue **javac** y que se encuentra explicado con detalle en el subapartado “Funcionamiento de **javac**”, como el número de la ronda, la lista de ficheros de entrada (“input files”), los tipos anotación detectados para ser procesados (“annotations”), si es la última ronda o no (“last round”) los procesadores encontrados en el proceso de descubrimiento, su emparejamiento con los tipos anotación, así como otra información que puede ser de gran relevancia a la hora de depurar el comportamiento de nuestro procesador de anotaciones.

Además de las opciones anteriores, puede también añadirse la opción **-verbose** para una salida aún más detallada de todo el proceso, esta vez incluyendo también información sobre las acciones de análisis y compilación de ficheros de código fuente.

Una vez invocado **javac**, se llevará a cabo el procesamiento de **@ImprimirMensaje**, obteniendo por la salida del compilador la información y mensajes de los elementos de código anotados. A continuación sigue un extracto de la salida del compilador:

NOTA: La información de depuración aparece en color verde y, cuando se ha obviado la salida por ser demasiado larga se indica con una frase entre puntos suspensivos ... y **en negrita**.

```
Round 1:
    input files: {...ficheros de código Java sobre los que se ha invocado el procesamiento...}
    annotations: [anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.ImprimirMensaje
                  ...y otros tipos anotación encontrados en los ficheros de código (si los
hay) ...]
        last round: false
Processor anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.ImprimirMensajeProcessor
matches [anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.ImprimirMensaje] and returns
false.
Note: procesador:
anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.ImprimirMensajeProcessor
Note: procesador: init: inicializando procesador...
Note: procesador: init: finalizado.
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion =
[anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.ImprimirMensaje]
Note: procesador: process: roundEnv = [errorRaised=false, rootElements=[...todas las clases...], processingOver=false]
Note: procesador: process: ronda final = false
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java67\src\main\java\anotaciones\procesamiento\java67\ejemplos\imprimirMensaje\elementosAnotados\Im-
primirMensajeClaseAnotada.java:14: Note: CLASE ANOTADA
public class ImprimirMensajeClaseAnotada {
    ^
Note: CLASE:
anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.elementosAnotados.ImprimirMensajeClaseA-
notada
Note: MENSAJE: Definicion de la clase.

... muchos mensajes más ...

D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java67\src\main\java\anotaciones\procesamiento\java67\ejemplos\imprimirMensaje\elementosAnotados\pa-
ckage-info.java:4: Note: PAQUETE ANOTADO
@ImprimirMensaje("Paquete java67.ejemplos.imprimirMensaje.elementosAnotados")
 ^
Note: PAQUETE:anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.elementosAnotados
Note: MENSAJE: Paquete java67.ejemplos.imprimirMensaje.elementosAnotados
Round 2:
    input files: {}
    annotations: []
    last round: true
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion = []
Note: procesador: process: roundEnv = [errorRaised=false, rootElements=[], processingOver=true]
Note: procesador: process: ronda final = true
Note: procesador:
anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.ImprimirMensajeProcessor finalizado.
```

Como se ve, hay una primera ronda en la que se toman en cuenta todos los ficheros Java especificados, se buscan sus tipos anotación, se hace el emparejamiento del procesador de **@ImprimirMensaje** con su tipo anotación y se ejecuta, imprimiendo los mensajes deseados. Finalmente, se llama de nuevo al procesador en la ronda 2 (nótese que sin tipos anotación a procesar). La ronda 2 la “ronda final” y, por tanto, ahí concluye el procesamiento.

Con este último paso, finalmente hemos concluido todas las fases del procesamiento de anotaciones. En los siguientes apartados expondremos más ejemplos de tipos anotación con requisitos más complejos y que cubren la diversidad de aplicaciones más habituales cuando se trabaja con procesamiento de anotaciones, como son la validación o la generación de código.

15.5.- Ejemplos: Procesadores JSR 269.

En este apartado se describen diversos ejemplos para profundizar en todas las etapas de implementación del procesamiento de anotaciones JSR 269. Se ofrecen varios ejemplos para las típicas aplicaciones del procesamiento de anotaciones: (1) navegación e información sobre el código fuente, (2) validación y (3) generación de ficheros. Para cada una de estas aplicaciones se ofrece un ejemplo básico y otro más avanzado para poder profundizar algo más en el tema.

NOTA: El nivel de código de los ejemplos es para Java SE 7 para incluir el procesamiento de las anotaciones sobre los nuevos elementos introducidos en Java SE 7

15.5.1.- Navegación básica: @ImprimirMensaje.

Se trata del ejemplo que hemos utilizado en el apartado anterior para explicar paso a paso las diferentes fases de las que se compone todo el proceso de procesamiento de anotaciones.

Requisitos:

Diseñar un tipo anotación que permita imprimir por la salida del compilador un mensaje textual asociado a cualquier tipo de elemento de código del lenguaje Java.

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface ImprimirMensaje {

    String value() default "¡Hola, mundo! (mensaje por defecto)";
}
```

Implementación:

Proyecto: **anotaciones-proc-java67**.

Paquete: **anotaciones.procesamiento.java67.ejemplos.imprimirMensaje**.

Clases anotadas:

Las clases anotadas, al ser muchas y para que no se mezclaran con las clases que implementan el procesamiento, se han alojado en el subpaquete llamado **elementosAnotados**.

Invocación:

```
javac -version -source 7 -proc:only -Xlint:-options ^
-XprintProcessorInfo -XprintRounds ^
-cp .\target\classes ^
-s .\src\main\java-generated ^
-processor anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.ImprimirMensajeProcessor ^
.\src\main\java\anotaciones\procesamiento\java67\ejemplos\imprimirMensaje\elementosAnotados\*.java
```

Resultados:

Una vez invocado correctamente **javac** y realizado el procesamiento de anotaciones por parte del procesador del tipo anotación, obtenemos por la salida del compilador la información de los elementos de código anotados y sus mensajes, como hemos visto en el apartado anterior.

15.5.2.- Navegación avanzada: @Nota.

En este ejemplo vamos a hacer un uso más avanzado de la navegación por la jerarquía de elementos usando un Visitor para recorrer la jerarquía de los mismos. Además, introduciremos más requisitos de funcionalidad sobre el tipo anotación a diseñar, así como otras características avanzadas, como la inclusión de opciones para el procesador de anotaciones.

Finalmente, la presentación de resultados, a diferencia de los ejemplos realizados hasta ahora, se realizará en la llamada que se hará al procesador en la “ronda final” del procesamiento.

Requisitos:

Diseñar un tipo anotación que imprima por la salida del compilador, en el orden más similar posible al código fuente, notas de texto asociadas a cualquier tipo de elemento del lenguaje Java. El orden puede diferir del orden exacto del código fuente en caso de definirse clases anidadas.

Para ofrecer la posibilidad de extender la funcionalidad del tipo anotación, estas notas se podrán calificar en tipos: **NOTA** (por defecto), **DOCUMENTACION**, **TAREA**, **PENDIENTE**, **AVISO** y **ERROR**. Las notas de tipo **AVISO** y **ERROR** generarán un warning y un error respectivamente a la salida del compilador. Esto no tiene mucha utilidad práctica, si no que se incluye a efectos ilustrativos. Se deberán ofrecer opciones para incluir o excluir específicamente una lista de cualquiera de estos tipos de notas. Por defecto: incluir **TODO**, excluir **NADA**.

Se podrá establecer un valor de severidad de la nota respectiva relacionada con la importancia de su contenido: **NORMAL** (por defecto), **LEVE**, **GRAVE** y **FATAL**. Se deberán ofrecer opciones para incluir o excluir específicamente una lista de cualquiera de estas severidades. Por defecto: incluir **TODO**, excluir **NADA**.

Para permitir ordenar, si se desea, las diferentes notas atendiendo a diferentes criterios, el tipo anotación incluirá un campo de prioridad de tipo entero, donde la prioridad irá, en principio, de mayor a menor, por lo que la mayor prioridad vendrá dada por **Integer.MAX_VALUE**, la menor por **Integer.MIN_VALUE** y el valor por defecto será **0**. No obstante, deberá poderse configurar mediante las opciones pertinentes los rangos que queremos seleccionar (por defecto: el rango completo), así como si se ordenará de forma ascendente (por defecto) o descendente.

Opciones soportadas:

Opciones soportadas por el procesador del tipo anotación @Nota		
Opción	Valores válidos	Por defecto
-Atipo.nota.incluir -Atipo.nota.excluir	Lista de tipos de notas separados por comas. Valores especiales: TODO y NADA . Si se encuentra alguno de estos valores especiales, se ignoran los demás.	Incluir TODO . Excluir NADA .
-Aseveridad.incluir -Aseveridad.excluir	Lista de tipos de severidad separados por comas. Valores especiales: TODO y NADA . Si se encuentra alguno de estos valores especiales, se ignoran los demás.	Incluir TODO . Excluir NADA .
-Aprioridad.minima -Aprioridad.maxima	Rango de prioridad mínima y máxima. Las notas que queden fuera de este rango no serán procesadas.	Mín: Integer.MIN_VALUE Máx: Integer.MAX_VALUE
-Aorden.tipo	Tipo de ordenación de las notas de entre las disponibles: CODIGO (por orden de aparición en código fuente, por defecto), TIPO , SEVERIDAD , PRIORIDAD .	CODIGO
-Aorden.direccion	Dirección de la ordenación de las notas: ASC (ascendente, por defecto) y DESC (descendente).	ASC

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Nota {

    // =====
    // ELEMENTOS
    // =====

    String texto() default "n/a";
    TipoNota tipo() default TipoNota.NOTA;
    SeveridadNota severidad() default SeveridadNota.NORMAL;
    int prioridad() default 0;

    // =====
    // TIPOS ENUMERADOS ESTÁTICOS
    // =====

    public static enum TipoNota { NOTA, DOCUMENTACION, TAREA, PENDIENTE, AVISO, ERROR };
    public static enum SeveridadNota { NORMAL, LEVE, GRAVE, FATAL };

    public static enum TipoOrden { CODIGO, TIPO, SEVERIDAD, PRIORIDAD };
    public static enum DireccionOrden { ASC, DESC };
}
```

Implementación:

Proyecto: **anotaciones-proc-java67**.

Paquete: **anotaciones.procesamiento.java67.ejemplos.nota**.

Clases anotadas:

Se incluye una clase anotada llamada **NotaClaseAnotada** de ejemplo en el mismo paquete.

Invocación:

La invocación de este ejemplo incluye, además de las opciones no estándar de depuración, que nos pueden venir bien para ver detalles del procesamiento en sí, todas las opciones soportadas por el procesador del tipo anotación **@Nota**, por lo que la invocación sale bastante aparatoso:

```
javac -version -source 7 -proc:only -Xlint:-options ^
-XprintProcessorInfo -XprintRounds ^
-cp .\target\classes ^
-s .\src\main\java-generated ^
-processor anotaciones.procesamiento.java67.ejemplos.nota.NotaProcessor ^
-Atipo.nota.incluir=NOTA,DOCUMENTACION,AVISO,TAREA,PENDIENTE,ERROR ^
-Atipo.nota.excluir=NADA ^
-Aseveridad.incluir=NORMAL,LEVE,GRAVE,FATAL ^
-Aseveridad.excluir=NADA ^
-Aprioridad.minima=-1234 ^
-Aprioridad.maxima=1234 ^
-Aorden.tipo=CODIGO ^
-Aorden.direccion=ASC ^
.\src\main\java\anotaciones\procesamiento\java67\ejemplos\nota\*.java
```

Resultados:

Una vez invocado **javac** y realizado el procesamiento, obtenemos por la salida del compilador las notas de los elementos anotados que se ajustan a las opciones de tipos de notas, severidades y prioridades especificadas, ordenadas según el orden especificado a través de las opciones de configuración del procesador de anotaciones.

Para este ejemplo, se han añadido líneas de información adicionales que informan de las etapas de procesamiento que va ejecutando el procesador: inicialización, procesamiento de las opciones de configuración, recuperación de los elementos raíz sobre los que descubrir los elementos anotados a procesar y, finalmente, el procesamiento, ordenación y escritura de resultados.

Para la invocación dada, se obtienen los siguientes resultados:

NOTA: La información de depuración aparece en **color verde** y, cuando se ha obviado la salida por ser demasiado larga se indica con una frase entre puntos suspensivos ... y **en negrita**.

```
Round 1:
    input files: {...ficheros de código Java sobre los que se ha invocado el procesamiento...}
    annotations: [anotaciones.procesamiento.java67.ejemplos.nota.Nota
                  ...y otros tipos anotación encontrados en los ficheros de código (si los
hay)...]
        last round: false
Processor anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.ImprimirMensajeProcessor
matches [anotaciones.procesamiento.java67.ejemplos.imprimirMensaje.ImprimirMensaje] and returns
false.
Processor anotaciones.procesamiento.java67.ejemplos.nota.NotaProcessor matches
[anotaciones.procesamiento.java67.ejemplos.nota.Nota] and returns false.
Note: procesador: anotaciones.procesamiento.java67.ejemplos.nota.NotaProcessor
Note: procesador: init: inicializando procesador...
Note: procesador: init: finalizado.
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion = [anotaciones.procesamiento.java67.ejemplos.nota.Nota]
Note: procesador: process: roundEnv = [errorRaised=false, rootElements=[...todas las clases...],
processingOver=false]
Note: procesador: process: ronda final = false
Note: procesador: opciones: parseando opciones...
Note: tiposNotasAProcesar = [NOTA, DOCUMENTACION, TAREA, PENDIENTE, AVISO, ERROR]
Note: severidadesAProcesar = [NORMAL, LEVE, GRAVE, FATAL]
Note: prioridadMinimaAProcesar = -1234
Note: prioridadMaximaAProcesar = 1234
Note: tipoOrdenAProcesar = CODIGO
Note: direccionOrdenAProcesar = ASC
Note: procesador: opciones: opciones parseadas.
Note: procesador: elementos raiz: recuperando elementos raiz...
Note: procesador: elementos raiz: 4 elementos raiz recuperados.
Note: procesador: procesamiento: iniciando procesamiento...
Note: procesador: procesamiento: procesamiento completado.
Round 2:
    input files: {}
    annotations: []
    last round: true
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion = []
Note: procesador: process: roundEnv = [errorRaised=false, rootElements=[], processingOver=true]
Note: procesador: process: ronda final = true
Note: procesador: ordenacion: ordenando resultados...
Note: procesador: ordenacion: resultados ordenados.
```

```

Note: procesador: resultados: escribiendo resultados...
Note: @Nota: Texto: Definición de la clase NotaClaseAnotada
      Tipo: NOTA, Severidad: NORMAL, Prioridad: 1
Note: @Nota: Texto: Tipo enumerado de la clase NotaClaseAnotada - PENDIENTE de añadir nuevos
valores
      Tipo: PENDIENTE, Severidad: LEVE, Prioridad: 333
Note: @Nota: Texto: valorEnum1 del tipo enumerado de la clase NotaClaseAnotada
      Tipo: PENDIENTE, Severidad: LEVE, Prioridad: 444
Note: @Nota: Texto: valorEnum2 del tipo enumerado de la clase NotaClaseAnotada
      Tipo: PENDIENTE, Severidad: LEVE, Prioridad: 555
warning: @Nota: Texto: Variable de clase de NotaClaseAnotada - AVISO
      Tipo: AVISO, Severidad: GRAVE, Prioridad: 111
Note: @Nota: Texto: Esta variable de instancia bla bla bla - DOCUMENTACION
      Tipo: DOCUMENTACION, Severidad: NORMAL, Prioridad: 222
Note: @Nota: Texto: Constructor de la clase NotaClaseAnotada
      Tipo: NOTA, Severidad: NORMAL, Prioridad: 666
Note: @Nota: Texto: Mútodo imprimirMensaje de la clase NotaClaseAnotada - TAREA de añadir un flujo
al método para no escribir directamente a System.out
      Tipo: TAREA, Severidad: FATAL, Prioridad: 777
Note: @Nota: Texto: Parámetro "mensaje" del mÚtodo imprimirMensaje de la clase NotaClaseAnotada
      Tipo: DOCUMENTACION, Severidad: NORMAL, Prioridad: 888
warning: @Nota: Texto: Variable local del método imprimirMensaje de la clase NotaClaseAnotada
      Tipo: AVISO, Severidad: GRAVE, Prioridad: 999
Note: @Nota: Texto: Definición del paquete anotaciones.procesamiento.java67.ejemplos.nota
      Tipo: DOCUMENTACION, Severidad: NORMAL, Prioridad: 0
Note: procesador: resultados: resultados escritos.
Note: procesador: anotaciones.procesamiento.java67.ejemplos.nota.NotaProcessor finalizado.
2 warnings

```

Comentarios:

La salida del compilador es la esperada. Se muestran todos los avisos incluidos según las opciones definidas para el procesador. Las dos notas de tipo **AVISO** se muestran como warnings. Si hubiéramos anotado algún elemento con una nota de tipo **ERROR** se generaría un error de compilación. Generar errores de compilación no tiene mucho sentido en la práctica, pero se ha incluido e efectos ilustrativos de una de las capacidades más interesantes de los procesadores: la validación, funcionalidad esta a la que le dedicaremos los siguientes ejemplos.

Las notas no se muestran exactamente como están escritas en el código fuente, ya que las notas del tipo anidado enumerado definido en la clase **NotaClaseAnotada** se muestran antes debido a que con el **ElementScanner** se recorre la jerarquía de subelementos en post-order y no exactamente en el orden en que los elementos aparecen escritos en el código fuente.

Además, y dado que hemos utilizado la Compiler Tree API para descubrir las anotaciones sobre variables locales, se muestra la nota correspondiente a la variable local interna al método `imprimirMensaje` con prioridad 999.

Por supuesto, se puede invocar el procesamiento cambiando las opciones de procesamiento para incluir y excluir ciertos tipos de notas u ordenarlas según los diferentes criterios que se deseen. Todo siguiendo los requisitos especificados. Emplazamos al lector a probar con diferentes combinaciones de opciones, comprobando los resultados y la ordenación de los mismos.

En la implementación de este ejemplo avanzado de navegación se ha tratado de ceñirse a las mejores prácticas recomendadas de programación. Para ello, se han definido tipos enumerados para modelar las características únicas del tipo anotación **@Nota**, ya que estos favorecen la posibilidad de incluir nuevos valores. Además, se han definido las cadenas que modelan las opciones del procesador como constantes de la clase procesador. Usar enumerados y constantes está recomendado para facilitar el mantenimiento y refactorización del código fuente de forma sencilla en caso de que hubiera que efectuar modificaciones sobre los requisitos a implementar.

Lo siguiente es comentar brevemente la estructura del procesamiento implementado para este procesador. Lo primero es comprobar si estamos en la ronda final y, si es así, presentar los resultados. Si no es así, se recuperan los elementos raíz de la ronda y se recorre su jerarquía de subelementos con un Visitor de tipo **ElementScanner**. El Visitor se encargará de conseguir que se llame el método auxiliar **procesarElemento** para todas las anotaciones descubiertas, incluidas las anotaciones sobre variables locales, para lo cual utilizamos la Compiler Tree API.

Aquí sigue un esquema simplificado del método **process** del código fuente del ejemplo:

```

@Override
public boolean process(Set<? extends TypeElement> tiposAnotacion, RoundEnvironment roundEnv) {

    // PROCESAMIENTO SEGÚN LA RONDA EN LA QUE NOS ENCONTREMOS

    if (roundEnv.processingOver()) {

        // PROCESAMIENTO DE LA RONDA FINAL -> PRESENTACIÓN DE RESULTADOS

        // presentamos los resultados del procesamiento
        this.presentarResultados();

    } else {

        // PROCESAMIENTO DE LOS ELEMENTOS ANOTADOS

        // OPCIONES DEL PROCESADOR

        // parseamos y procesamos las opciones del procesador
        this.parsearYProcesarOpcionesProcesador();

        // LISTA DE ELEMENTOS RESULTADO

        // inicializamos la lista de elementos que se incluirán en el resultado
        elementosResultado = new ArrayList<Element>();

        // PROCESAMIENTO DE ANOTACIONES DE LOS ELEMENTOS RAÍZ

        // recuperamos la lista de elementos raíz a procesar por este procesador
        Set<? extends Element> elementosRaiz = roundEnv.getRootElements();

        // instanciamos un Visitor de tipo ElementScanner para
        // rastrear la jerarquía completa de elementos y subelementos
        NotaElementScanner notaElementScanner = new NotaElementScanner();

        // pasamos el ElementScanner sobre todos los elementos raíz
        for (Element elemento : elementosRaiz) {

            // pasamos el ElementScanner sobre el elemento
            notaElementScanner.scan(elemento);
        }

        // NOTA: No se hace nada más en process. Todo el procesamiento se acabará
        // haciendo en el método auxiliar procesarElemento(Element elemento), que
        // será el que llame el scanner para todos los elementos y subelementos
        // de los elementos raíz. En procesarElemento guardaremos la información
        // de las anotaciones procesadas y, en la ronda final, presentaremos todos
        // los resultados filtrados y ordenados según las opciones del procesador.
    }

    // finalizamos retornando false porque no queremos quedarnos con el procesado
    // de este tipo anotación en exclusiva, si devolvemos true el conjunto de
    // anotaciones que hemos procesado no será pasado a otros procesadores
    return false;
} // process

```

El método **procesarElemento** realiza el procesamiento de los elementos visitados; en este caso, es filtrarlos según el método **esElementoAProcesar** para ver si sus notas son elegibles para ser incluidas en el resultado, según las opciones especificadas para el procesador en la invocación:

```
private void procesarElemento(Element elem) {  
    // PROCESAMIENTO DEELEMENTOS  
  
    // comprobamos si el elemento debe incluirse en el resultado  
    if (esElementoAProcesar(elem)) {  
  
        // incluimos el elemento en la lista resultado  
        // (si no ha sido añadida previamente)  
        if (!elementosResultado.contains(elem)) {  
            elementosResultado.add(elem);  
        }  
    }  
  
    private boolean esElementoAProcesar(Element elem) {  
  
        // nos quedamos con los elementos anotados por @Nota  
        // que además sean del tipo, severidad y prioridad configuradas  
        boolean esElementoAProcesar = true;  
  
        // recuperamos la anotación en cuestión  
        Nota anotacionNota = elem.getAnnotation(claseAnotacionAProcesar);  
  
        // comprobamos que cumpla todas las condiciones para ser  
        // seleccionado para mostrarse en el resultado:  
  
        // ¿está anotado con @Nota?  
        esElementoAProcesar =  
            esElementoAProcesar && (anotacionNota != null);  
  
        // ¿su tipo está entre los tipos de notas a incluir?  
        esElementoAProcesar =  
            esElementoAProcesar && tiposNotasAProcesar.contains(anotacionNota.tipo());  
  
        // ¿su severidad está entre las severidades a incluir?  
        esElementoAProcesar =  
            esElementoAProcesar && severidadesAProcesar.contains(anotacionNota.severidad());  
  
        // ¿su prioridad se encuentra en el rango a procesar?  
        esElementoAProcesar =  
            esElementoAProcesar  
            && anotacionNota.prioridad() >= prioridadMinimaAProcesar  
            && anotacionNota.prioridad() <= prioridadMaximaAProcesar;  
  
        // finalmente, devolvemos el resultado  
        return esElementoAProcesar;  
    }  
}
```

Finalmente, tras completarse la ronda inicial de procesamiento, se alcanza la ronda final y se presentan los resultados a través del método **presentarResultados**.

NOTA: Teniendo los elementos resultado en una colección, ya filtradas y ordenadas según los criterios especificados en las opciones del procesador, podríamos ampliar la funcionalidad de nuestro procesador añadiendo como requisito, por ejemplo, volcar dicha información a un fichero de texto (usando `createTextFile` la interfaz **Filer**) o incluso alimentar la creación de informes en PDF, HTML, XLS, CSV, XML, etc, mediante la utilización de alguna API Java de informes (reporting) como [JasperReports](#). Esto se deja como ejercicio práctico para el lector.

Consideraciones especiales sobre el ElementScanner del ejemplo

Es interesante comentar la implementación del `ElementScanner` utilizado para rastrear la jerarquía de los elementos y sus respectivos subelementos para descubrir elementos anotados.

Lo más interesante del `ElementScanner` que se ha definido es (1) **cómo se implementa el procesamiento de los tipos de elementos para lograr descubrir variables locales** utilizando métodos que se sirven de las facilidades de la Compiler Tree API. Además, también hay que resaltar que (2) **IMPORTANTE: se ha redefinido el método visitUnknown**, algo que deberá tenerse en cuenta siempre que se utilice un Visitor de la API JSR 269 si queremos que nuestros procesadores no provoquen problemas en futuras versiones de Java.

Se recomienda al lector que examine el código completo de NotaElementScanner leyendo y tratando de comprender todos los comentarios del mismo.

Lo siguiente es una lista de las **principales consideraciones sobre NotaElementScanner**:

- **Todos los métodos:** En todos los casos, se llama `procesarElemento` sobre el elemento visitado antes de visitar los subelementos. Esto a efectos prácticos provoca que se recorra el árbol de subelementos visitados en [pre-orden](#) (primero la raíz y luego los nodos hijos).
- **visitType:** Después del procesamiento, para los tipos se llama a `procesarInicializadores` para que se procesen las variables locales anotadas dentro de los bloques inicializadores, tanto estáticos como de instancia, que puedan definir las clases o tipos enumerados que se visiten.
- **visitExecutable:** Tras el procesamiento, para elementos ejecutables (métodos, constructores y bloques inicializadores) se llama a `procesarVariablesElementoEjecutable` para que se procesen todas las variables contenidas en los cuerpos de dichos elementos.
- **visitTypeParameter:** Su código es igual que los otros métodos sencillos, pero se incluye una nota que comenta que, dado que los parámetros de tipo no son anotables en Java SE 7, este método nunca será invocado. No obstante, se deja la misma implementación que a los demás tipos de elementos porque este procesador podría ser invocado para procesar código cliente escrito con un nivel de código Java SE 8 o superior y para este código el compilador ya sí soporta anotaciones sobre tipos parámetros. En ese caso, como querremos que se cumplan los requisitos del procesamiento, nos interesa dejar el cuerpo del método como el de cualquier otro elemento.
- **visitUnknown:** Este método se redefine porque su implementación original arroja una excepción `UnknownElementException` y no queremos que esto ocurra. Dado que la lógica de procesamiento del procesador de nuestro ejemplo no tiene en cuenta el tipo de elemento a la hora de procesarlo, podemos limitarnos a procesar los elementos desconocidos de la misma manera que los demás. Para mantener el contrato de los métodos de `ElementScanner` (que dice que deben visitarse el elemento actual así como todos sus subelementos), se pasa el scanner a todos los subelementos del elemento desconocido. **IMPORTANTE:** Independientemente de que la lógica del procesador de anotaciones tenga en cuenta el tipo de elemento o no, siempre será una buena política redefinir el método `visitUnknown`, ya sea con un cuerpo vacío o uno que emita algún mensaje o warning, si no queremos que dicho procesador corra el riesgo de arrojar excepciones y fastidiar la compilación cuando se introduzcan nuevos tipos de elementos en futuras versiones de la plataforma Java.

Aquí se deja el código de **NotaElementScanner** para que el lector pueda examinarlo:

```
private class NotaElementScanner extends ElementScanner7<Void, Void> {

    // =====
    // MÉTODOS DE APLICACIÓN
    // =====

    @Override
    public Void visitPackage(PackageElement pqElem, Void param) {

        // procesamos el elemento
        procesarElemento(pqElem);

        // llamamos al método de la superclase para que se siga llevando
        // a cabo la lógica de llamar a los subelementos de este elemento
        // (al hacerlo después de haber procesado el elemento, estaremos
        // procesando el árbol de jerarquía de los elementos en pre-order)
        super.visitPackage(pqElem, param);

        // devolvemos como resultado el parámetro
        // sólo para conformar la firma del método
        return param;
    }

    @Override
    public Void visitType(TypeElement tipoElem, Void param) {

        // procesamos el elemento
        procesarElemento(tipoElem);

        // procesamos las variables dentro del elemento tipo visitado
        procesarInicializadores(tipoElem);

        // llamamos al método de la superclase para que se siga llevando
        // a cabo la lógica de llamar a los subelementos de este elemento
        // (al hacerlo después de haber procesado el elemento, estaremos
        // procesando el árbol de jerarquía de los elementos en pre-order)
        super.visitType(tipoElem, param);

        // devolvemos como resultado el parámetro
        // sólo para conformar la firma del método
        return param;
    }

    @Override
    public Void visitExecutable(ExecutableElement ejecutableElem, Void param) {

        // procesamos el elemento
        procesarElemento(ejecutableElem);

        // procesamos las variables dentro del elemento ejecutable visitado
        procesarVariablesElementoEjecutable(ejecutableElem);

        // llamamos al método de la superclase para que se siga llevando
        // a cabo la lógica de llamar a los subelementos de este elemento
        // (al hacerlo después de haber procesado el elemento, estaremos
        // procesando el árbol de jerarquía de los elementos en pre-order)
        super.visitExecutable(ejecutableElem, param);

        // devolvemos como resultado el parámetro
        // sólo para conformar la firma del método
        return param;
    }
}
```

```

@Override
public Void visitVariable(VariableElement vrbleElem, Void param) {
    // ... mismo cuerpo que visitPackage ...
}

@Override
public Void visitTypeParameter(TypeParameterElement paramTipoElem, Void param) {
    // procesamos el elemento
    procesarElemento(paramTipoElem);

    // IMPORTANTE: Los parámetros de tipo no son anotables en Java SE 7,
    // con lo que, en principio, no tiene sentido procesar los elementos
    // de este tipo, pero como en Java SE 8 sí es posible anotar los
    // parámetros de tipo, seguimos dejándolo, ya que este procesador
    // podría procesar código compilado en Java SE 8 y, en ese caso,
    // querremos que exhiba el comportamiento dado por los requisitos.

    // llamamos al método de la superclase para que se siga llevando
    // a cabo la lógica de llamar a los subelementos de este elemento
    // (al hacerlo después de haber procesado el elemento, estaremos
    // procesando el árbol de jerarquía de los elementos en pre-order)
    super.visitTypeParameter(paramTipoElem, param);

    // devolvemos como resultado el parámetro
    // sólo para conformar la firma del método
    return param;
}

@Override
public Void visitUnknown(Element elemTipoDesconocido, Void param) {
    // procesamos el elemento
    procesarElemento(elemTipoDesconocido);

    // IMPORTANTE: Redefinimos visitUnknown para que, en caso de que
    // este Visitor se encontrara con un tipo de elemento desconocido
    // NO se realice la implementación por defecto, que es arrojar una
    // UnknownElementException (no queremos arrojar ninguna excepción).
    // Para este procesador, podemos redefinir de forma segura este
    // método porque EL TIPO CONCRETO DEL ELEMENTO ES IRRELEVANTE PARA
    // LA IMPLEMENTACIÓN DE NUESTRA LÓGICA, ya que las anotaciones del
    // tipo anotación @Nota se comportan igual sea cual sea el tipo
    // del elemento que estén anotando: paquete, clase, interfaz, etc.
    // Para mantener el contrato de los ElementScanner, que es visitar
    // a los subelementos de los elementos visitados, en este caso
    // NO podemos llamar a super.visitUnknown(elemTipoDesconocido, param),
    // ya que arrojaría la excepción. Lo que haremos será llamar
    // recursivamente a nuestra versión redefinida de visitUnknown
    // sobre los subelementos del elemento que estamos visitando:
    // (al hacerlo después de haber procesado el elemento, estaremos
    // procesando el árbol de jerarquía de los elementos en pre-order)
    for (Element subelemento : elemTipoDesconocido.getEnclosedElements()) {
        this.visitUnknown(subelemento, param);
    }

    // devolvemos como resultado el parámetro
    // sólo para conformar la firma del método
    return param;
}
}

```

15.5.3.- Validación básica: @Contexto.

En el presente ejemplo introduciremos una de las funcionalidades más habituales para las que se utiliza el procesamiento de anotaciones: la **validación de código fuente**.

Al disponer de las facilidades para navegar por los elementos del código fuente en tiempo de compilación, poder evaluar sus diferentes propiedades y emitir errores de compilación desde el entorno del procesador de anotaciones, ello nos permitirá implementar procesadores que se encarguen de validar que un cierto código fuente cumpla unas ciertas condiciones.

Requisitos:

Diseñar un tipo anotación aplicable sobre campos que permita comprobar si el tipo declarado de dicho campo pertenece a una implementación de una interfaz de contexto de una aplicación llamada, por ejemplo, **IContextoApp**.

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.FIELD)
public @interface Contexto { }
```

Implementación:

Proyecto: **anotaciones-proc-java67**.

Paquete: **anotaciones.procesamiento.java67.ejemplos.contexto**.

Clase anotada:

Se incluye la clase anotada **ContextoClaseAnotada** como ejemplo en el mismo paquete.

Invocación:

```
javac -version -proc:only -Xlint:-options ^
-XprintProcessorInfo -XprintRounds ^
-cp .\target\classes ^
-s .\src\main\java-generated ^
-processor anotaciones.procesamiento.java67.ejemplos.contexto.ContextoProcessor ^
.\src\main\java\anotaciones\procesamiento\java67\ejemplos\contexto\*.java
```

Resultados:

Al invocar el procesamiento y procesarse la clase anotada se obtienen los resultados esperados: los elementos anotados con `@Contexto` sobre campos cuyo tipo declarado o no implementa `IContextoApp` o es una interfaz provoca un error de compilación, mientras que los campos anotados con `@Contexto` sobre clases que implementan la interfaz `IContextoApp` no dan ningún problema. En concreto en la clase `ContextoClaseAnotada` se deben encontrar 7 declaraciones anotadas, como se ve en la salida del procesador, de las cuales 5 arrojan errores. Al iniciar la primera ronda de procesamiento también se arroja un warning debido a que el nivel de código fuente soportado por este procesador es `SourceVersion.RELEASE_6` mientras que el del compilador `javac` en nuestro caso es 7, pero esto no supone ningún problema en absoluto.

NOTA: La información de depuración aparece en color verde y, cuando se ha obviado la salida por ser demasiado larga se indica con una frase entre puntos suspensivos ... y **en negrita**.

```
Round 1:
    input files: {...ficheros de código Java sobre los que se ha invocado el procesamiento...}
    annotations: [anotaciones.procesamiento.java67.ejemplos.contexto.Contexto
                  ...y otros tipos anotación encontrados en los ficheros de código (si los
hay) ...]
        last round: false
Processor anotaciones.procesamiento.java67.ejemplos.contexto.ContextoProcessor matches
[anotaciones.procesamiento.java67.ejemplos.contexto.Contexto] and returns false.
Note: procesador: anotaciones.procesamiento.java67.ejemplos.contexto.ContextoProcessor
Note: procesador: init: inicializando procesador...
Note: procesador: init: finalizado.
warning: Supported source version 'RELEASE_6' from annotation processor
'anotaciones.procesamiento.java67.ejemplos.contexto.ContextoProcessor' less than -source '1.7'
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion =
[anotaciones.procesamiento.java67.ejemplos.contexto.Contexto]
Note: procesador: process: roundEnv = [errorRaised=false, rootElements=[...todas las clases...],
processingOver=false]
Note: procesador: process: ronda final = false
Note: procesador: elementos raiz: recuperando elementos anotados...
Note: procesador: elementos raiz: 7 elementos raiz recuperados.
Note: procesador: procesamiento: iniciando procesamiento...
error: tipo declarado del campo valorEnumNoContexto
(anotaciones.procesamiento.java67.ejemplos.contexto.ContextoClaseAnotada.TipoEnumerado) no
implementa la interfaz anotaciones.procesamiento.java67.ejemplos.contexto.IContextoApp
error: tipo declarado del campo variableClaseContextoInterfaz
(anotaciones.procesamiento.java67.ejemplos.contexto.IContextoApp) no implementa la interfaz
anotaciones.procesamiento.java67.ejemplos.contexto.IContextoApp
error: tipo declarado del campo variableClaseNoContexto (java.lang.String) no implementa la
interfaz anotaciones.procesamiento.java67.ejemplos.contexto.IContextoApp
error: tipo declarado del campo variableInstanciaContextoInterfaz
(anotaciones.procesamiento.java67.ejemplos.contexto.IContextoApp) no implementa la interfaz
anotaciones.procesamiento.java67.ejemplos.contexto.IContextoApp
error: tipo declarado del campo variableInstanciaNoContexto (java.lang.Integer) no implementa la
interfaz anotaciones.procesamiento.java67.ejemplos.contexto.IContextoApp
Note: procesador: procesamiento: procesamiento completado.
Round 2:
    input files: {}
    annotations: []
    last round: true
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion = []
Note: procesador: process: roundEnv = [errorRaised=true, rootElements=[], processingOver=true]
Note: procesador: process: ronda final = true
Note: procesador: anotaciones.procesamiento.java67.ejemplos.contexto.ContextoProcessor finalizado.
5 errors
```

Comentarios:

El procesador `@Contexto` es muy sencillo gracias a que la validación a realizar es local a cada campo, por lo que es posible simplemente recuperar todos los elementos anotados con `getElementsAnnotatedWith` y procesarlos iterando directamente sobre ellos sin ningún orden en particular. Para cada uno se comprueba que su tipo no sea una interfaz y que sea assignable a `IContextoApp`, es decir: que el tipo del campo sea una implementación de dicha interfaz.

```
@Override
public boolean process(Set<? extends TypeElement> tiposAnotacion, RoundEnvironment roundEnv) {
    // PROCESAMIENTO SEGÚN LA RONDA EN LA QUE NOS ENCONTRAMOS

    if (roundEnv.processingOver()) {

        // PROCESAMIENTO DE LA RONDA FINAL

        // mensaje informativo para mostrar que el procesador ha finalizado su procesamiento
        this.message.printMessage(Kind.NOTE, "procesador: " +
            this.getClass().getCanonicalName() + " finalizado.");

    } else {

        // PROCESAMIENTO DE ANOTACIONES DE LOS ELEMENTOS RAÍZ

        // recuperamos la lista de elementos anotados a procesar por este procesador
        Set<? extends Element> elementosAnotados =
            roundEnv.getElementsAnnotatedWith(Contexto.class);

        // procesamos cada elemento anotado
        for (Element elementoAnotado : elementosAnotados) {

            // PROCESAMIENTO DE LA DECLARACIÓN

            // sabemos que la anotación sólo se puede usar sobre campos -> down-cast
            VariableElement campoElem = (VariableElement) elementoAnotado;

            // TipeMirror del campo
            TypeMirror campoTypeMirror = campoElem.asType();

            // tipo declarado del campo
            TypeElement tipoCampoTypeElement =
                this.elements.getTypeElement(campoTypeMirror.toString());

            // comprobamos si el tipo del campo no es en sí mismo una interfaz
            // y es assignable a (es decir: que implementa) la interfaz IContextoApp
            boolean anotacionValida =
                !(tipoCampoTypeElement.getKind() == ElementKind.INTERFACE)
                && this.types.isAssignable(campoTypeMirror, this.tipoIContextoApp);

            // si no es una anotación válida, tenemos que emitir un error de compilación
            if (!anotacionValida) {
                this.message.printMessage(Kind.ERROR,
                    "tipo declarado del campo " + campoElem.getSimpleName()
                    + " (" + campoTypeMirror + ")"
                    + " no implementa la interfaz " + this.tipoIContextoApp);
            }
        }
    }

    // finalizamos retornando false porque no queremos quedarnos con el procesado
    // de este tipo anotación en exclusiva, si devolvemos true el conjunto de
    // anotaciones que hemos procesado no será pasado a otros procesadores
    return false;
} // process
```

15.5.4.- Validación avanzada: @JavaBean.

En el siguiente ejemplo vamos a profundizar en la de validación de código fuente haciendo que esta no afecte sólo a elementos de código aislados, si no que las condiciones a validar afectarán al conjunto de toda la clase y requerirá de una mayor complejidad.

Requisitos:

Diseñar un tipo anotación aplicable exclusivamente sobre clases que permita comprobar si la clase anotada es un [Java Bean](#), es decir, que cumple con los siguientes requisitos:

- 1) La clase es serializable (es decir, que implementa la interfaz `java.io.Serializable`).
- 2) La clase tiene un constructor sin argumentos que permite crear instancias sin información.
- 3) Todos los campos de la clase son accesibles a través de métodos setter y getter. Contemplar que los métodos getter de los campos tipo `boolean` puedan empezar alternativamente por “`i s`”.

En caso de que no se cumpla cualquiera de estos requisitos, el procesamiento de la anotación `@JavaBean` deberá emitir un error de compilación vinculado a la declaración de clase anotada.

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface JavaBean { }
```

Implementación:

Proyecto: `anotaciones-proc-java67`.

Paquete: `anotaciones.procesamiento.java67.ejemplos.javaBean`.

Clase anotada:

Se incluyen las clases anotadas `JavaBeanClaseAnotadaJavaBean` como ejemplo de clase Java Bean válida y `JavaBeanClaseAnotadaNoJavaBean` como ejemplo de clase Java que cumple algunos de los requisitos para ser Java Bean, pero no llega a serlo.

Invocación:

```
javac -version -proc:only -Xlint:-options ^
-XprintProcessorInfo -XprintRounds ^
-cp .\target\classes ^
-s .\src\main\java-generated ^
-processor anotaciones.procesamiento.java67.ejemplos.javaBean.JavaBeanProcessor ^
.\src\main\java\anotaciones\procesamiento\java67\ejemplos\javaBean\*.java
```

Resultados:

Cuando se invoca el procesamiento de anotaciones sobre el código del paquete **javaBean**, los resultados son los esperados: se encuentran 2 elementos de clase anotados. Una las clases cumple los requisitos para ser Java Bean, y la otra no y el procesador arroja un error:

NOTA: La información de depuración aparece en **color verde** y, cuando se ha obviado la salida por ser demasiado larga se indica con una frase entre puntos suspensivos ... y **en negrita**.

Round 1:

```
    input files: {...ficheros de código Java sobre los que se ha invocado el procesamiento...}
    annotations: [anotaciones.procesamiento.java67.ejemplos.javaBean.JavaBean
                  ...y otros tipos anotación encontrados en los ficheros de código (si los
hay) ...]
        last round: false
Processor anotaciones.procesamiento.java67.ejemplos.javaBean.JavaBeanProcessor matches
[anotaciones.procesamiento.java67.ejemplos.javaBean.JavaBean] and returns false.
Note: procesador: anotaciones.procesamiento.java67.ejemplos.javaBean.JavaBeanProcessor
Note: procesador: init: inicializando procesador...
Note: procesador: init: finalizado.
warning: Supported source version 'RELEASE_6' from annotation processor
'anotaciones.procesamiento.java67.ejemplos.javaBean.JavaBeanProcessor' less than -source '1.7'
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion =
[anotaciones.procesamiento.java67.ejemplos.javaBean.JavaBean]
Note: procesador: process: roundEnv = [errorRaised=false, rootElements=[...todas las clases...],
processingOver=false]
Note: procesador: process: ronda final = false
Note: procesador: elementos anotados: recuperando elementos anotados...
Note: procesador: elementos anotados: 2 elementos anotados recuperados.
Note: procesador: procesamiento: iniciando procesamiento...
D:\PFC\Contenido\3-Desarrollo\EclipseWorkspace\anotaciones-proc-
java67\src\main\java\anotaciones\procesamiento\java67\ejemplos\javaBean\JavaBeanClaseAnotadaNoJavaB
ean.java:15: error: @JavaBean: error: la clase
anotaciones.procesamiento.java67.ejemplos.javaBean.JavaBeanClaseAnotadaNoJavaBean no tiene setter
y/o getter para los siguientes campos: [variableInstanciaInteger, variableInstanciaBoolean]
public class JavaBeanClaseAnotadaNoJavaBean implements java.io.Serializable {
    ^
Note: procesador: procesamiento: procesamiento completado.
```

Round 2:

```
    input files: {}
    annotations: []
        last round: true
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion = []
Note: procesador: process: roundEnv = [errorRaised=true, rootElements=[], processingOver=true]
Note: procesador: process: ronda final = true
Note: procesador: anotaciones.procesamiento.java67.ejemplos.javaBean.JavaBeanProcessor finalizado.
1 warning
```

Comentarios:

En el código del **JavaBeanProcessor**, ya que el tipo anotación **@JavaBean** sólo se aplica a clases y nos da igual el orden, hemos utilizado el método **getElementsAnnotatedWith** para traernos directamente todos los elementos anotados en un orden indefinido.

Después, para cada elemento anotado, debemos comprobar que sea una clase y no una interfaz o un tipo enumerado. **@JavaBean** es **@Target(ElementType.TYPE)** (incluye clases, interfaces y enumerados) y los usuarios podrían haber colocado la anotación incorrectamente sobre interfaces o tipos enumerados, en cuyo caso deberá arrojarse el correspondiente error. Finalmente, pasamos a comprobar los requisitos exigidos para considerar la clase Java Bean. Lo siguiente es un fragmento del método **process** del procesador: el cuerpo del bucle **for** donde se realiza el procesamiento para cada uno de los elementos anotados:

```
// procesamos cada elemento anotado
for (Element elementoAnotado : elementosAnotados) {

    // PROCESAMIENTO DE LOS ELEMENTOS ANOTADOS

    // sabemos que la anotación sólo se puede usar sobre tipos,
    // así que podemos hacer un down-cast de forma segura
    TypeElement tipoElem = (TypeElement) elementoAnotado;

    // COMPROBACIÓN PREVIA: EL ELEMENTO NO ES OTRA COSA QUE UNA CLASE

    // debido a que un TypeElement puede corresponder a una interfaz
    // (incluso a un tipo anotación) o a un tipo enumerado, hay que comprobarlo

    if (tipoElem.getKind() != ElementKind.CLASS) {

        // la anotación @JavaBean se ha puesto sobre una interfaz o un tipo enumerado:
        // arrojamos un error de compilación
        this.messager.printMessage(Kind.ERROR,
            "@JavaBean: error: @JavaBean sólo es aplicable"
            + " a clases (" + elementoAnotado.getSimpleName() + " no es una clase)",
            elementoAnotado);

        // y pasamos a procesar el siguiente elemento anotado
        continue;
    }

    // COMPROBACIÓN DE LOS REQUISITOS JAVA BEAN SOBRE EL ELEMENTO ANOTADO DE LA CLASE

    // comprobamos que el elemento anotado de clase cumple los requisitos para ser Java Bean:
    // 1) La clase es serializable (es decir, que implementa la interfaz java.io.Serializable).
    // 2) La clase tiene un constructor sin argumentos para crear instancias sin información.
    // 3) Todos los campos de la clase son accesibles a través de sus métodos setter y getter.

    // COMPROBACIÓN 1) SERIALIZABLE

    // ¿implementa la clase la interfaz java.io.Serializable?
    if (!this.types.isAssignable(tipoElem.asType(), this.tipoInterfazSerializable)) {

        // la clase no implementa la interfaz java.io.Serializable -> ERROR
        this.messager.printMessage(Kind.ERROR,
            "@JavaBean: error: la clase " + tipoElem
            + " no implementa java.io.Serializable",
            tipoElem);
    }
}
```

```

// COMPROBACIÓN 2) CONSTRUCTOR SIN ARGUMENTOS

// comprobamos si la clase no tiene el constructor sin argumentos que requerimos
if (!claseTieneConstructorSinArgumentos(tipoElem)) {

    // entre la lista de constructores no está el constructor sin argumentos -> ERROR
    this.messager.printMessage(Kind.ERROR,
        "@JavaBean: error: la clase " + tipoElem
        + " no tiene un constructor sin argumentos",
        tipoElem);
}

// COMPROBACIÓN 3) CAMPOS ACCESIBLES A TRAVÉS DE MÉTODOS SETTER Y GETTER

// obtenemos la lista de campos de la clase
List<VariableElement> campos = ElementFilter.fieldsIn(tipoElem.getEnclosedElements());

// comprobamos si hay campos que no tengan correctamente definidos su setter y getter
ArrayList<VariableElement> camposSinSetterOGetter = new ArrayList<VariableElement>();

for (VariableElement campoElem : campos) {

    // ¿carece el campo de método setter o de método getter?
    if (!campoTieneMetodoSetter(tipoElem, campoElem)
        || !campoTieneMetodoGetter(tipoElem, campoElem)) {

        // el campo carece de método setter o getter
        // -> lo añadimos a la lista de campos sin setter o getter
        camposSinSetterOGetter.add(campoElem);
    }
}

// ¿hay campos que carezcan de métodos setter o getter?
if (!camposSinSetterOGetter.isEmpty()) {

    // si hay campos que carecen de métodos setter o getter -> ERROR
    this.messager.printMessage(Kind.ERROR,
        "@JavaBean: error: la clase " + tipoElem
        + " no tiene setter y/o getter para "
        + "los siguientes campos: " + camposSinSetterOGetter,
        tipoElem);
}

} // for Element

```

Como podemos ver, el código anterior está estructurado en 3 bloques de no mucho tamaño, gracias a la refactorización de las operaciones más aparatosas a otros 3 métodos auxiliares: `claseTieneConstructorSinArgumentos`, `campoTieneMetodoSetter` y `campoTieneMetodoGetter`.

A continuación se incluye el método más interesante de entre los método auxiliares citados: **campoTieneMetodoSetter**. En él se hacen comprobaciones de condiciones y comprobaciones sobre tipos. **claseTieneConstructorSinArgumentos** tiene una implementación muy sencilla que simplemente itera sobre los constructores de la clase para comprobar si existe el constructor sin argumentos. **campoTieneMetodoGetter** tiene como peculiaridad el hecho que contempla la posibilidad de que el getter de una propiedad booleana pueda empezar por “is”, no obstante, a parte de eso, es muy similar en su construcción a **campoTieneMetodoSetter**.

```

private boolean campoTieneMetodoSetter(
    TypeElement claseElem,
    final VariableElement campoElem) {

    // vrble resultado
    boolean campoTieneMetodoSetter = false;

    // nombre del método setter correspondiente al campo
    String nombreCampo = campoElem.getSimpleName().toString();
    final String nombreSetter = "set"
        + nombreCampo.substring(0, 1).toUpperCase()
        + nombreCampo.substring(1);

    // recuperamos los métodos de la clase
    List<ExecutableElement> metodos =
        ElementFilter.methodsIn(claseElem.getEnclosedElements());

    // vrbles auxiliares
    boolean esMetodoSetterCampo = false;
    TypeMirror tipoVoid = this.types.getNoType(TypeKind.VOID);
    TypeMirror tipoCampo = campoElem.asType();

    // iteramos sobre los métodos para ver si encontramos
    // que uno de ellos es el método setter del campo
    for (ExecutableElement metodo : metodos) {

        // ¿es el método actual el setter del campo?
        // condiciones a cumplir:
        // 1) que sea un método (no constructor) public
        // 2) que su tipo de retorno sea void
        // 3) que el nombre del método sea el esperado para el setter
        // 4) que tenga 1 parámetro
        // 5) que el tipo del parámetro sea assignable al del campo
        esMetodoSetterCampo =
            metodo.getKind() == ElementKind.METHOD
            && metodo.getModifiers().contains(Modifier.PUBLIC)
            && this.types.isSameType(metodo.getReturnType(), tipoVoid)
            && metodo.getSimpleName().toString().equals(nombreSetter)
            && metodo.getParameters().size() == 1
            && this.types.isAssignable(metodo.getParameters().get(0).asType(), tipoCampo);

        // si hemos encontrado el método setter del campo
        // -> ponemos el resultado a true y salimos del for
        if (esMetodoSetterCampo) {
            campoTieneMetodoSetter = true;
            break;
        }
    }

    // devolvemos el resultado
    return campoTieneMetodoSetter;
}

```

15.5.5.- Generación de código básica: @Proxy.

En el presente ejemplo introduciremos la última de las tipologías de funcionalidad más importantes del procesamiento de anotaciones: la **generación de ficheros**.

La generación de ficheros, especialmente de nuevos ficheros de código fuente, fue una de las necesidades más importantes que vino a cubrir la introducción del procesamiento de anotaciones en el lenguaje Java y en las que más se pensó en un principio. Actualmente, no obstante, cada vez están más en boga las aplicaciones de los procesadores de anotaciones como validadores adicionales de semánticas complejas en tiempo de compilación.

Para la generación de nuevos ficheros de código fuente durante el procesamiento de anotaciones, la API de procesamiento JSR 269 nos proporciona la interfaz **Filer** con métodos para crear ficheros Java de código o de clase, así como ficheros de recurso.

Requisitos:

Diseñar un tipo anotación aplicable exclusivamente sobre clases que permita la creación de una subclase que añadirá estado adicional (en forma de una nueva variable de instancia), así como comportamiento adicional (añadiendo la implementación de una nueva interfaz).

Más concretamente, deben de cumplirse las siguientes condiciones:

- 1) El nombre de la nueva subclase será el nombre de la clase original más el sufijo “**Proxy**”.
- 2) El paquete de la nueva subclase será también el de la clase original más el sufijo “**Proxy**”.
- 3) La subclase añadirá una variable de tipo **EstadoProxy** e implementará la interfaz **IProxy**.

Tipo anotación diseñado:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface Proxy { }
```

Implementación:

Proyecto: **anotaciones-proc-java67**.

Paquete: **anotaciones.procesamiento.java67.ejemplos.proxy**.

Clase anotada:

Se incluye la clase anotada **ProxyClaseAnotada** que generará a **ProxyClaseAnotadaProxy** dentro del paquete **anotaciones.procesamiento.java67.ejemplos.proxyProxy**.

Invocación:

```
javac -version -Xlint:-options ^
-XprintProcessorInfo -XprintRounds ^
-cp .\target\classes ^
-s .\src\main\java-generated ^
-d .\target\classes ^
-processor anotaciones.procesamiento.java67.ejemplos.proxy.ProxyProcessor ^
.\src\main\java\anotaciones\procesamiento\java67\ejemplos\proxy\*.java
```

Resultados:

Cuando se invoca el procesamiento sobre el código del paquete **proxy**, se encuentra un elemento de clase anotado, el correspondiente a la clase **ProxyClaseAnotada**, y se genera el fichero correspondiente a su subclase, llamado **ProxyClaseAnotadaProxy** dentro del paquete **anotaciones.procesamiento.java67.ejemplos.proxy** en la ruta especificada por la opción **-s** de la invocación, en este caso: **-s .\src\main\java-generated**. En la invocación de los ejemplos anteriores, casi como si fuera una plantilla, habíamos especificado siempre el valor de la opción **-s**, pero, ya que en ellos no se generaban nuevos ficheros, especificar esa opción no servía a efectos prácticos para nada. En este ejemplo es donde tiene verdadero sentido especificar dicha opción y se utiliza efectivamente por primera vez.

Finalmente, como en esta ocasión no hemos especificado la opción **-proc:only**, también se compilará la nueva clase generada, guardándose su fichero de clase compilado **.class** en el directorio dado por **-d .\target\classes**.

Lo más interesante del resultado es que, al haber generado nuevo código fuente, **javac** entra en una segunda ronda de procesamiento utilizando como entrada el nuevo fichero generado y finalmente llega a la ronda 3 que ya sí es la ronda final:

NOTA: La información de depuración aparece en **color verde** y, cuando se ha obviado la salida por ser demasiado larga se indica con una frase entre puntos suspensivos ... y **en negrita**.

```
Round 1:
  input files: {...ficheros de código Java sobre los que se ha invocado el procesamiento...}
  annotations: [anotaciones.procesamiento.java67.ejemplos.proxy.Proxy
                ...y otros tipos anotación encontrados en los ficheros de código (si los
hay) ...]
    last round: false
Processor anotaciones.procesamiento.java67.ejemplos.proxy.ProxyProcessor matches
[anotaciones.procesamiento.java67.ejemplos.proxy.Proxy] and returns false.
Note: procesador: anotaciones.procesamiento.java67.ejemplos.proxy.ProxyProcessor
Note: procesador: init: inicializando procesador...
Note: procesador: init: finalizado.
warning: Supported source version 'RELEASE_6' from annotation processor
'anotaciones.procesamiento.java67.ejemplos.proxy.ProxyProcessor' less than -source '1.7'
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion = [anotaciones.procesamiento.java67.ejemplos.proxy.Proxy]
Note: procesador: process: roundEnv = [errorRaised=false, rootElements=[...todas las clases...],
processingOver=false]
Note: procesador: process: ronda final = false
Note: procesador: elementos anotados: recuperando elementos anotados...
Note: procesador: elementos anotados: 1 elementos anotados recuperados.
Note: procesador: procesamiento: iniciando procesamiento...
Note: procesador: procesamiento: clase:
anotaciones.procesamiento.java67.ejemplos.proxy.ProxyClaseAnotada
Note: procesador: procesamiento: fichero: generando fichero...
Note: procesador: procesamiento: fichero: creando manejador fichero...
Note: procesador: procesamiento: fichero: manejador de fichero creado.
Note: procesador: procesamiento: fichero: abriendo fichero...
Note: procesador: procesamiento: fichero: fichero abierto.
Note: procesador: procesamiento: fichero: escribiendo fichero...
Note: procesador: procesamiento: fichero: fichero escrito.
Note: procesador: procesamiento: fichero: cerrando fichero...
Note: procesador: procesamiento: fichero: fichero cerrado.
Note: procesador: procesamiento: procesamiento completado.
```

```

Round 2:
    input files: {anotaciones.procesamiento.java67.ejemplos.proxyProxy.ProxyClaseAnotadaProxy}
    annotations: [javax.annotation.Generated]
    last round: false
Processor anotaciones.procesamiento.java67.ejemplos.proxy.ProxyProcessor matches [] and returns
false.
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion = []
Note: procesador: process: roundEnv = [errorRaised=false,
rootElements=[anotaciones.procesamiento.java67.ejemplos.proxyProxy.ProxyClaseAnotadaProxy],
processing
Over=false]
Note: procesador: process: ronda final = false
Note: procesador: elementos anotados: recuperando elementos anotados...
Note: procesador: elementos anotados: 0 elementos anotados recuperados.
Note: procesador: procesamiento: iniciando procesamiento...
Note: procesador: procesamiento: procesamiento completado.
Round 3:
    input files: {}
    annotations: []
    last round: true
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion = []
Note: procesador: process: roundEnv = [errorRaised=false, rootElements=[], processingOver=true]
Note: procesador: process: ronda final = true
Note: procesador: anotaciones.procesamiento.java67.ejemplos.proxy.ProxyProcessor finalizado.
1 warning

```

IMPORTANTE: Nótese cómo en la información de depuración acerca de la ronda 2 se refleja que se reciben como ficheros de entrada (“input files”) los nuevos ficheros de código generados en la ronda anterior, en este caso el fichero de la clase **ProxyClaseAnotadaProxy**. Dicha clase sólo se encuentra anotada por la Common Annotation `@Generated`. Nuestro **ProxyProcessor** no procesa dicho tipo de anotación, pero recordemos que **javac siempre invoca en las rondas subsiguientes todos los procesadores invocados en alguna de las rondas anteriores**. Así, en cuanto un procesador es invocado en una ronda, ya será invocado en todas las siguientes, incluyendo por supuesto la ronda final. Es por este hecho que **ProxyProcessor** también es invocado en la ronda 2 aunque no procese en dicha ronda ningún tipo anotación:

Processor anotaciones.procesamiento.java67.ejemplos.proxy.ProxyProcessor
matches [] and returns false.

IMPORTANTE: En la ronda 2 **ProxyProcessor** recupera 0 elementos anotados recuperados. Esto se debe a que la lógica del procesador, cuando no estamos en la ronda final, procesa los elementos anotados recuperados por `getElementsAnnotatedWith`. No obstante, en este caso, al no haber ningún elemento anotado, no se recuperan elementos anotados. Es por eso que en la ronda 2 **ProxyProcessor** realiza una “ronda de procesamiento vacío”, procesando 0 elementos anotados.

En la ronda 2, por tanto, se consume la clase generada **ProxyClaseAnotadaProxy** sin ser procesada por ningún procesador y, al no generarse nuevo código, en la ronda 3 ya entramos en la ronda final de procesamiento.

En la ronda final, el **ProxyProcessor** ejecuta la lógica implementada para dicha ronda final, imprimiendo simplemente un mensaje informativo de que ha finalizado.

Una vez finalizada la ejecución de todos los procesadores en la ronda final, se da por terminado el procesamiento de anotaciones y **javac**, a no ser que se especifique `-proc:only` (que no es el caso de la invocación de este ejemplo), **javac** compilará todos los ficheros de código, tanto los ficheros originales como los nuevos ficheros generados, guardándose todos los ficheros de clase compilados en la ruta dada por la opción `-d`, en este caso: `-d .\target\classes`.

Comentarios:

Al terminar el procesamiento del **ProxyProcessor**, vemos que se ha generado **ProxyClaseAnotadaProxy.java** en el directorio de salida de ficheros generados, dado por la opción **-s .\src\main\java-generated**.

El fichero **.java** ha sido compilado tras la ronda final de procesamiento y su **.class** compilado se encontrará en la ruta destino dada por la opción **-d .\target\classes**.

Si estuviéramos aún en fase de desarrollo del procesador, podríamos abrir el **.java** generado para comprobar si se está generando el código fuente exacto que queremos y así corregir posibles errores. Este es el fichero **.java** generado:

```
package anotaciones.procesamiento.java67.ejemplos.proxyProxy;

import javax.annotation.Generated;

import anotaciones.procesamiento.java67.ejemplos.proxy.*;

/**
 * Clase proxy generada para la clase
anotaciones.procesamiento.java67.ejemplos.proxy.ProxyClaseAnotada.
 * Amplia la funcionalidad de la clase original con los métodos de la interfaz <code>IProxy</code>.
 * Añade también la variable de estado para el funcionamiento del proxy <code>EstadoProxy</code>.
 */
@Generated(value = { "anotaciones.procesamiento.java67.ejemplos.proxy.ProxyProcessor" },
comments = "Procesador de anotaciones ejemplo @Proxy.", date = "2014-06-09T22:24+0200")
public class ProxyClaseAnotadaProxy extends ProxyClaseAnotada implements IProxy {

    // =====
    // VARIABLES DE INSTANCIA
    // =====

    // variable de estado para el proxy
    private EstadoProxy estadoProxy = new EstadoProxy();

    // =====
    // MÉTODOS DE APLICACIÓN
    // =====

    public void proxyMetodo1(String argumentoMetodo1) {

        // ... implementación del método 1 ...
        this.estadoProxy.setVrbleEstado1(argumentoMetodo1);
    }

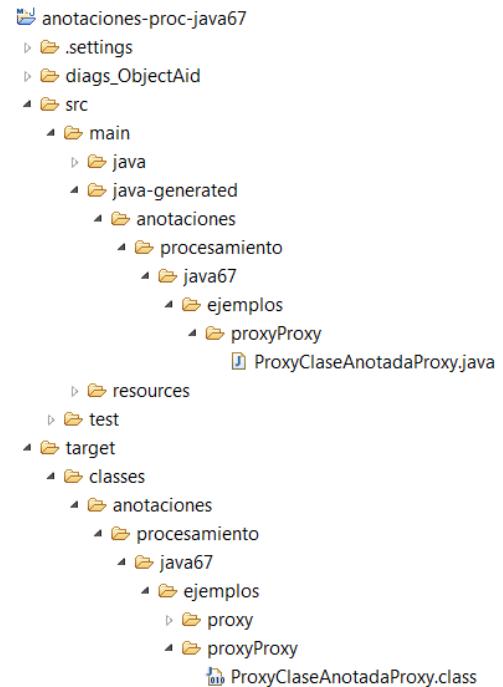
    public void proxyMetodo2(Integer argumentoMetodo2) {

        // ... implementación del método 2 ...
        this.estadoProxy.setVrbleEstado2(argumentoMetodo2);
    }

    public Float proxyMetodo3(Long argumentoMetodo3) {

        // ... implementación del método 3 ...
        this.estadoProxy.setVrbleEstado3(
            (float) (2 * Math.PI * argumentoMetodo3));

        // devolvemos el resultado
        return this.estadoProxy.getVrbleEstado3();
    }
}
```



En cuanto al código del **ProxyProcessor**, ya que el tipo anotación **@Proxy** sólo se aplica a clases y nos da igual el orden, hemos vuelto a usar el método **getElementsAnnotatedWith** para traernos directamente todos los elementos anotados en un orden indefinido.

Para cada elemento anotado, comprobamos que sea una clase y no una interfaz o un tipo enumerado, ya que **@Proxy** es **@Target(ElementType.TYPE)** (incluye clases, interfaces y enumerados) y los usuarios podrían colocar la anotación incorrectamente sobre interfaces o tipos enumerados, caso en el cual el procesador deberá arrojar el correspondiente error.

A continuación, preparamos la escritura del código fuente creando una serie de variables que nos vendrán bien y que facilitarán la legibilidad del código del procesador. Estas variables tendrán las propiedades tanto de la clase original como de la clase nueva a generar:

```
// PREPARACIÓN DEL FICHERO DE LA NUEVA SUBCLASE

// en este punto, ya sabemos que tenemos un elemento de clase, hacemos down-cast
TypeElement claseElem = (TypeElement) elementoAnotado;

this.messageer.printMessage(Kind.NOTE,
    "procesador: procesamiento: clase: " + claseElem.getQualifiedName());

// propiedades de la clase original
PackageElement pqElem = (PackageElement) claseElem.getEnclosingElement();
String paqueteClaseOriginal = pqElem.getQualifiedName().toString();
String nombreClaseOriginal = claseElem.getSimpleName().toString();
String nombreCanonicoClaseOriginal = paqueteClaseOriginal + "." + nombreClaseOriginal;

// propiedades de la nueva clase a generar
String paqueteNuevaClase = pqElem.getQualifiedName().toString() + "Proxy";
String nombreNuevaClase = claseElem.getSimpleName().toString() + "Proxy";
String nombreCanonicoNuevaClase = paqueteNuevaClase + "." + nombreNuevaClase;
```

El paso siguiente es utilizar la interfaz **Filer** para crear el manejador del nuevo fichero de código Java a crear. Este manejador será una instancia de la interfaz **JavaFileObject**, que es la que modela los ficheros de código Java dentro de Java Compiler API y la API JSR 269. De dicho **JavaFileObject** obtendremos un flujo escritor de caracteres **java.io.Writer** a través de su método **openWriter**, que decoraremos con un **PrintWriter** para poder escribir el código del nuevo fichero generado más cómodamente con su método **println**.

La creación y escritura del fichero está rodeada de un bloque **try-catch** para capturar una **IOException** que pudiera ocurrir durante la generación del nuevo fichero de código:

```
// creamos el manejador del nuevo fichero...
JavaFileObject fichero = null;
PrintWriter ficheroWriter = null;

try {
    // CREACIÓN DEL FICHERO

    // creamos un manejador de fichero asociado al nombre canónico de la nueva clase
    this.messager.printMessage(Kind.NOTE,
        "procesador: procesamiento: fichero: creando manejador fichero...");
    fichero = this.filer.createSourceFile(nombreCanonicoNuevaClase);
    this.messager.printMessage(Kind.NOTE,
        "procesador: procesamiento: fichero: manejador de fichero creado.");

    // creamos el flujo escritor del nuevo fichero
    this.messager.printMessage(Kind.NOTE,
        "procesador: procesamiento: fichero: abriendo fichero...");
    ficheroWriter = new PrintWriter(fichero.openWriter());
    this.messager.printMessage(Kind.NOTE,
        "procesador: procesamiento: fichero: fichero abierto.");

    // CÓDIGO FUENTE DE LA NUEVA SUBCLASE
    // ... escritura del código a fichero (ver página siguiente) ...

} catch (IOException e) {

    // ERROR DURANTE LA CREACIÓN DEL FICHERO

    // error durante el proceso de creación del
    // fichero de la nueva clase -> ERROR
    this.messager.printMessage(Kind.ERROR,
        "@Proxy: error: error durante la "
        + "generación del fichero correspondiente a la "
        + "clase " + claseElem.getQualifiedName() + " -> " + e);

} finally {

    // CIERRE DEL FICHERO

    // cerramos el flujo del fichero
    this.messager.printMessage(Kind.NOTE,
        "procesador: procesamiento: fichero: cerrando fichero...");
    if (ficheroWriter!=null) ficheroWriter.close();
    this.messager.printMessage(Kind.NOTE,
        "procesador: procesamiento: fichero: fichero cerrado.");
}
```

Dentro del **try-catch**, tras la apertura del fichero, sólo resta escribir el código fuente de la clase que queremos generar al flujo de salida del fichero. Cuando terminemos, el fichero se cerrará con la llamada al método **close** del bloque **finally**.

Todo el cuerpo de escritura del código de la nueva clase es un poco monótono y aparatoso, ya que consta de una llamada tras otra al método de escritura `ficheroWriter.println` (nótese que siguiendo la convención establecida, hemos anotado el código generado con `@Generated`):

```

// CÓDIGO FUENTE DE LA NUEVA SUBCLASE

this.messager.printMessage(Kind.NOTE,
    "procesador: procesamiento: fichero: escribiendo fichero...");

// paquete
ficheroWriter.println("package " + paqueteNuevaClase + ";" + "\n");

// imports
ficheroWriter.println("import javax.annotation.Generated;" + "\n"); // para la anotación @Generated
ficheroWriter.println("import " + paqueteClaseOriginal + ".*;" + "\n");

// cabecera javadoc de la nueva clase
ficheroWriter.println("/**");
ficheroWriter.println("* Clase proxy generada para la clase " + nombreCanonicoClaseOriginal + ".");
ficheroWriter.println("* Amplia la funcionalidad de la clase original con los métodos de la");
ficheroWriter.println("* interfaz <code>IProxy</code>.");
ficheroWriter.println("* Añade también la variable de estado para el funcionamiento del proxy");
ficheroWriter.println("<code>EstadoProxy</code>.");
ficheroWriter.println("*/");

// anotación @Generated (nueva de Java SE 6) sobre la definición de la clase;
// valores de los elementos (establecidos según las convenciones de la
// documentación oficial de Java SE 6+ sobre el tipo anotación @Generated):
// value: nombre canónico de la clase generadora de código
// comments: nombre de la utilidad de generación de código o cualquier otro comentario libre
// date: fecha de la generación de código en formato ISO 8601
ficheroWriter.println("@Generated("
    + "value = { \\" + this.getClass().getCanonicalName() + "\\}, " + "\n"
    + "comments = \\" + "Procesador de anotaciones ejemplo @Proxy." + "\", "
    + "date = \\" + formatoFechaISO8601.format(new Date()) + "\\)");
}

// definición de la nueva clase
ficheroWriter.println("public class " + nombreNuevaClase + " extends " + nombreClaseOriginal + " implements IProxy"
+ " ");

// bloque de variables de instancia
ficheroWriter.println("    ");
ficheroWriter.println("    // =====");
ficheroWriter.println("    // VARIABLES DE INSTANCIA");
ficheroWriter.println("    // =====");
ficheroWriter.println("    ");

// variables de instancia: la variable de estado del proxy
ficheroWriter.println("    // variable de estado para el proxy");
ficheroWriter.println("    private EstadoProxy estadoProxy = new EstadoProxy();");

// bloque de métodos de aplicación
ficheroWriter.println("    ");
ficheroWriter.println("    // =====");
ficheroWriter.println("    // MÉTODOS DE APLICACIÓN");
ficheroWriter.println("    // =====");
ficheroWriter.println("    ");

// método: proxyMetodo1
ficheroWriter.println("    public void proxyMetodo1(String argumentoMetodo1) {" );
ficheroWriter.println("        ");
ficheroWriter.println("        // ... implementación del método 1 ...");
ficheroWriter.println("        this.estadoProxy.setVrbleEstado1(argumentoMetodo1);");
ficheroWriter.println("    }");
ficheroWriter.println("    ");

```

```

// método: proxyMetodo2
ficheroWriter.println("    public void proxyMetodo2(Integer argumentoMetodo2) {" );
ficheroWriter.println("        ");
ficheroWriter.println("        // ... implementación del método 2 ...");
ficheroWriter.println("        this.estadoProxy.setVrbleEstado2(argumentoMetodo2);");
ficheroWriter.println("    }");
ficheroWriter.println("    ");

// método: proxyMetodo3
ficheroWriter.println("    public Float proxyMetodo3(Long argumentoMetodo3) {" );
ficheroWriter.println("        ");
ficheroWriter.println("        // ... implementación del método 3 ...");
ficheroWriter.println("        this.estadoProxy.setVrbleEstado3(");
ficheroWriter.println("            (float) (2 * Math.PI * argumentoMetodo3));");
ficheroWriter.println("        ");
ficheroWriter.println("        // devolvemos el resultado");
ficheroWriter.println("        return this.estadoProxy.getVrbleEstado3();");
ficheroWriter.println("    }");
ficheroWriter.println("    ");

// cerramos el cuerpo de la clase
ficheroWriter.println("}");

this.messager.printMessage(Kind.NOTE,
    "procesador: procesamiento: fichero: fichero escrito.");

```

Como vemos, la generación del nuevo código fuente del fichero no es especialmente complicada. Sin embargo, suele ser habitual que deban escribirse códigos a veces de muy gran tamaño, monótonos, llenos de llamadas al método `println`, y muy farragosos. Además de generar una estructura muy pobre de código fuente, son difíciles de editar y mantener.

Para mejorar la estructura de este código podría refactorizarse la escritura del código fuente a un método auxiliar que se encargara de la escritura a fichero y tratar de sacar el texto del código fuente a escribir a una variable `String` o a un fichero de plantilla.

La utilización de plantillas en este tipo de procesadores que implementan una funcionalidad de generación de código fuente es algo muy interesante a contemplar. Si la generación de código es a pequeña escala, puede ser más sencillo escribir directamente a fichero o crear una pequeña solución de reemplazo de variables sobre cadenas de caracteres.

No obstante, si necesitamos generar código fuente a gran escala, con varios tipos de ficheros a generar y/o de gran tamaño, existen multitud de [APIs de plantillas](#) que, a través de un motor de plantillas ([template engine](#)), permiten una cómoda generación de ficheros a través de la combinación de plantillas estáticas con datos o parámetros dinámicos. Algunas de las APIs de plantillas más conocidas y populares en la comunidad Java son [FreeMarker](#) o [Apache Velocity](#).

No vamos a profundizar más a este respecto, ya que el uso de motores de plantillas es algo más allá del ámbito de este manual, pero podría ser un ejercicio práctico interesante para el lector modificar el código del ejemplo para generar el mismo código utilizando alguna API de generación de plantillas de código fuente. En la siguiente URL de la tercera entrega de un excelente tutorial sobre procesamiento de anotaciones (incluido también en el subapartado “Tutoriales” de la “Bibliografía” del presente manual), se explica, por ejemplo, cómo utilizar una plantilla de [Apache Velocity](#) para generar el código de los nuevos ficheros de código fuente:

Code Generation using Annotation Processors in the Java language – part 3: Generating Source Code
<http://deors.wordpress.com/2011/10/31/annotation-generators/>

15.5.6.- Generación de código avanzada: @AutoTest.

En el presente ejemplo profundizaremos en la generación de ficheros de código fuente. En esta ocasión, la generación de código no será tan estática, si no que tendrá un componente más dinámico, pudiendo ocurrir que las clases generadas tengan un número de métodos variable, así como cuerpos diferentes.

Requisitos:

Se quiere implementar una pequeña API de **generación automática de tests de unidad** que, a diferencia de otras como [JUnit](#), generará los tests automáticamente a partir de los casos de prueba indicados por un conjunto de anotaciones. Para ello, se deberá diseñar un tipo anotación **@AutoTest**, aplicable exclusivamente sobre métodos de clases, que permitirá definir casos de prueba sencillos para cada método mediante anotaciones **@AutoTestCasoPrueba**.

En el tipo anotación **@AutoTestCasoPrueba** se indicará la lista de argumentos del método, tal y como se escribirían en código Java, separados por la barra vertical “|”, así como el resultado concreto que se espera para dicho test. Se presentan adicionalmente los siguientes requisitos:

- 1) El nombre de la nueva subclase será el de la clase original más el sufijo “**AutoTest**”.
- 2) El paquete de la nueva subclase será el de la clase original más el sufijo “**AutoTest**”.
- 3) La nueva subclase, como es convención habitual en Java, se anotará con **@Generated**.
- 4) Los auto tests se generarán agrupados por clases en el orden dado por su nombre canónico.
- 5) Los auto tests deberán poder probar métodos de resultados variables, **void** o errores.
- 6) Deberá especificarse al menos un **@AutoTestCasoPrueba** o el procesador dará un error.
- 7) Los tests se anotarán con **@AutoTestGenerado**, un tipo anotación con retención **RUNTIME**, para poder ser descubiertos y ejecutados en tiempo de ejecución por un “test runner”.
- 8) El procesador informará por la salida del compilador de los auto tests según sean generados.

Esta pequeña API de generación automática de tests supone una combinación interesante de las capacidades de generación de nuevos ficheros de código con los mecanismos de reflexión y otras características de las anotaciones, como, por ejemplo, las anotaciones múltiples de **@AutoTest** como tipo anotación contenedor y **@AutoTestCasoPrueba** como anotación contenida.

No obstante, este ejemplo lógicamente adolece de muchas restricciones. La principal de ella es la imposibilidad de pasar argumentos de clases complejas a través de los argumentos de las propias anotaciones. Esto se debe a las propias restricciones que el lenguaje Java establece sobre los valores posibles que puede tener el elemento de una anotación.

[JUnit](#), por ejemplo, solventa el problema de la instanciación y población de valor de los objetos de clases no triviales a través de su tipo anotación **@Before**, que anota el método de la clase de test que se encargará de construir en su cuerpo los objetos complejos que los tests necesitan para trabajar. Así pues, los métodos etiquetados con **@Before** se llaman justo antes de cada test de la clase para dejar el estado del test donde se deseé. No obstante, como hemos dicho, esta API no genera los tests de forma automática, así que dicho código tendrá que ser escrito por los desarrolladores. En el caso de este ejemplo, a efectos meramente ilustrativos, hemos tratado de llegar todo lo lejos que fuera posible sin que el usuario de nuestros tipos anotación tuviera que escribir una sola línea de código.

Tipos anotación diseñados:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.METHOD)
public @interface AutoTest {

    AutoTestCasoPrueba[] casosPrueba(); // implicitamente se toma: default { };

}

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.METHOD)
public @interface AutoTestCasoPrueba {

    // =====
    // ELEMENTOS
    // =====

    String argumentos() default "";
    String resultado() default "";
    AutoTestTipoResultado tipoResultado() default AutoTestTipoResultado.VARIABLE;

    // =====
    // TIPOS ENUMERADOS ESTÁTICOS
    // =====

    public enum AutoTestTipoResultado { VARIABLE, VOID, ERROR }
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AutoTestGenerado { }
```

Implementación:

Proyecto: **anotaciones-proc-java67**.

Paquete: **anotaciones.procesamiento.java67.ejemplos.autoTest**.

Clases anotadas:

Se han incluido las siguientes clases anotadas (dentro del paquete **autoTest**):

- **AutoTestClassAnotadaLecturaFicheros**
- **AutoTestClassAnotadaMultiplicarDividir**
- **AutoTestClassAnotadaSumarRestar**

Están listadas en el mismo orden lexicográfico ascendente en el que el procesador deberá procesarlas según los requisitos establecidos. Dichas clases generarán sus respectivas nuevas clases en **anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest**.

Invocación:

Para este ejemplo tendremos 2 invocaciones: la invocación habitual de **javac** para la ejecución del procesamiento de anotaciones sobre el tipo anotación **@AutoTest**, y otra adicional que llamará al comando **java** (el entorno de ejecución de la Máquina Virtual Java) para ejecutar el código de la clase **AutoTestRunner**, una especie de código ejecutor de los auto tests generados durante el procesamiento, para que sean ejecutados y comprobar sus resultados directamente.

```
javac -version -Xlint:-options ^
-XprintProcessorInfo -XprintRounds ^
-cp .\target\classes ^
-s .\src\main\java-generated ^
-d .\target\classes ^
-processor anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestProcessor ^
.\src\main\java\anotaciones\procesamiento\java67\ejemplos\autoTest\*.java

java ^
-cp ".\target\classes" ^
anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestRunner
```

Resultados:

En cuanto al procesamiento de anotaciones, obtenemos la salida esperada, listándose las clases en orden lexicográfico ascendente, así como los casos de prueba que se van generando para ellas. Lo más interesante, al igual que en el ejemplo anterior del tipo anotación **@Proxy**, será comprobar como, en esta ocasión también se alcanzan 3 rondas de procesamiento: la ronda 1 de generación de nuevos ficheros, la ronda 2 de procesamiento vacío ya que no hay procesadores para procesar los ficheros generados y la ronda 3 que es la ronda final.

A continuación sigue un resumen de la salida del procesador de anotaciones de **@AutoTest**, donde puede verse como en la ronda 1 se generan 3 nuevas “clases AutoTest” para las 3 clases anotadas con el tipo anotación **@AutoTest** en los cuerpos de sus métodos.

NOTA: La información de depuración aparece en color verde y, cuando se ha obviado la salida por ser demasiado larga se indica con una frase entre puntos suspensivos ... y **en negrita**.

```
Round 1:
    input files: {...ficheros de código Java sobre los que se ha invocado el procesamiento...}
    annotations: [anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTest
                  ...y otros tipos anotación encontrados en los ficheros de código (si los
hay)...]
        last round: false
Processor anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestProcessor matches
[anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTest] and returns false.
Note: procesador: anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestProcessor
Note: procesador: init: inicializando procesador...
Note: procesador: init: finalizado.
warning: Supported source version 'RELEASE_6' from annotation processor
'anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestProcessor' less than -source '1.7'
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion =
[anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTest]
Note: procesador: process: roundEnv = [errorRaised=false, rootElements=[...todas las clases...],
processingOver=false]
Note: procesador: process: ronda final = false
```

```

Note: procesador: elementos anotados: recuperando elementos anotados...
Note: procesador: elementos anotados: 6 elementos anotados recuperados.
Note: procesador: procesamiento: iniciando procesamiento...
Note: procesador: procesamiento: clase:
anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestClaseAnotadaLecturaFicheros
Note: procesador: procesamiento: fichero: generando fichero...
Note: procesador: procesamiento: fichero: creando manejador fichero...
Note: procesador: procesamiento: fichero: manejador de fichero creado.
Note: procesador: procesamiento: fichero: abriendo fichero...
Note: procesador: procesamiento: fichero: fichero abierto.
Note: procesador: procesamiento: fichero: escribiendo fichero...
Note: procesador: procesamiento: generando metodo de prueba: lecturaFicheroOK_casoPrueba_1
Note: procesador: procesamiento: generando metodo de prueba: lecturaFicheroError_casoPrueba_1
Note: procesador: procesamiento: fichero: fichero escrito.
Note: procesador: procesamiento: fichero: cerrando fichero...
Note: procesador: procesamiento: fichero: fichero cerrado.
Note: procesador: procesamiento: clase:
anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestClaseAnotadaMultiplicarDividir
... generación de la nueva subclases de métodos de prueba para la clase 2...
Note: procesador: procesamiento: clase:
anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestClaseAnotadaSumarRestar
... generación de la nueva subclases de métodos de prueba para la clase 3...
Note: procesador: procesamiento: procesamiento completado.
Round 2:
    input files:
{anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaLecturaFicherosAutoTest,
anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaMultiplicarDividirAutoTest,
anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaSumarRestarAutoTest}
    annotations: [javax.annotation.Generated,
anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestGenerado]
    last round: false
Processor anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestProcessor matches [] and
returns false.
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion = []
Note: procesador: process: roundEnv = [errorRaised=false,
rootElements=[anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaLecturaFicherosAutoTest,anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaMultiplicarDividirAutoTest,anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaSumarRestarAutoTest], processingOver=false]
Note: procesador: process: ronda final = false
Note: procesador: elementos anotados: recuperando elementos anotados...
Note: procesador: elementos anotados: 0 elementos anotados recuperados.
Note: procesador: procesamiento: iniciando procesamiento...
Note: procesador: procesamiento: procesamiento completado.
Round 3:
    input files: {}
    annotations: []
    last round: true
Note: procesador: process: iniciando ronda de procesamiento...
Note: procesador: process: tiposAnotacion = []
Note: procesador: process: roundEnv = [errorRaised=false, rootElements=[], processingOver=true]
Note: procesador: process: ronda final = true
Note: procesador: anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestProcessor finalizado.
1 warning

```

Como se ve, en la ronda 2 no se procesa ninguna de las nuevas clases generadas cuyos nombres terminan en “AutoTest”, ya que los tipos anotación **@AutoTestGenerado** y **@Generated** que están presentes en las nuevas clases generadas, no tienen ningún procesador que las reclame. No obstante, aún así, como sabemos, **javac** invoca todos los procesadores invocados en rondas anteriores en las rondas siguiente, por lo que se ejecuta de nuevo el **AutoTestProcessor** en una “ronda de procesamiento vacía”, en la que no empareja con ningún tipo anotación de los pendientes para la presente ronda (**matches []**) y donde, por tanto, no procesa nada. Como en la ronda 2 no se generan nuevos ficheros, se alcanza la ronda 3 como ronda final y se termina.

Al terminar el procesamiento de anotaciones, vemos que se han generado los 3 nuevos ficheros de código **.java** para las “clases AutoTest” en el directorio de salida de ficheros generados, dado por la opción **-s .\src\main\java-generated**.

Dichos ficheros **.java** serán compilados tras la ronda final de procesamiento y sus **.class** compilados se guardarán en la ruta destino dada por la opción **-d .\target\classes**.

Si estuviéramos aún en fase de desarrollo del procesador, podríamos abrir el **.java** generado para comprobar si se está generando el código fuente exacto que queremos y así corregir posibles errores. Este es uno de esos ficheros:

```
package anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest;
import javax.annotation.Generated;

import anotaciones.procesamiento.java67.ejemplos.autoTest.*;

/**
 * Clase de auto-tests generada a partir de los casos de prueba de la siguiente clase original:
 * anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestClaseAnotadaLecturaFicheros.
 */
@Generated(value = { "anotaciones.procesamiento.java67.ejemplos.autoTest.AutoTestProcessor" },
comments = "Procesador de anotaciones ejemplo @AutoTest.", date = "2014-06-10T11:36+0200")
public class AutoTestClaseAnotadaLecturaFicherosAutoTest {

    // =====
    // MÉTODOS DE PRUEBA
    // =====

    @AutoTestGenerado
    public boolean lecturaFicheroOK_casoPrueba_1() {

        try {

            // instancia del objeto a probar
            AutoTestClaseAnotadaLecturaFicheros objTest =
                    new AutoTestClaseAnotadaLecturaFicheros();

            // llamada con los argumentos especificados
            objTest.lecturaFicheroOK("C:\ruta\fichero\fichero_existente.txt");

            // presentación del resultado por consola
            System.out.println("AutoTest: lecturaFicheroOK_casoPrueba_1: metodo void ejecutado sin errores.");

            // comprobación de resultado: AUTO-TEST OK si no se arroja ninguna excepción
            return true;

        } catch (Throwable t) {

            // AUTO-TEST NOOK

            // presentación del error por consola
            System.out.println("AutoTest: lecturaFicheroOK_casoPrueba_1: error: " + t);

            return false;
        }
    }
}
```



```

@AutoTestGenerado
public boolean lecturaFicheroError_casoPrueba_1() {
    try {
        // instancia del objeto a probar
        AutoTestClaseAnotadaLecturaFicheros objTest =
            new AutoTestClaseAnotadaLecturaFicheros();

        // llamada con los argumentos especificados y recolección del resultado
        objTest.lecturaFicheroError("C:\ruta\fichero\fichero_no_existente.txt");

        // comprobación de resultado: AUTO-TEST NOOK si no hay errores
        return false;
    } catch (Throwable t) {
        // presentación del error por consola
        System.out.println("AutoTest: lecturaFicheroError_casoPrueba_1: error: " + t);

        // comprobación de resultado: AUTO-TEST OK
        // si se recibe una excepción del tipo esperado en el resultado
        return t.getClass().getCanonicalName().equals("java.io.FileNotFoundException");
    }
}
}

```

El tipo de resultado de los tests, tal y como se definió en sus requisitos, es de tres tipos posibles: **VARIABLE**, **VOID** y **ERROR**. Para ilustrar los métodos generados para los tipos **VOID** y **ERROR**, se ha definido la clase anotada `AutoTestClaseAnotadaLecturaFicheros`, con un método que no devuelve nada cuando se ejecuta (tipo de retorno **void**) y otro que está previsto que arroje un error al ejecutarse, ya que en el test se le pasará el valor de una ruta a un fichero inexistente. La “clase `AutoTest`” generada para dicha clase anotada es la dada por el código anterior (`AutoTestClaseAnotadaLecturaFicherosAutoTest`).

Para el tipo de resultado **VOID**, el código del método de prueba generado no utiliza e ignora el valor del elemento `resultado` de `@AutoTestCasoPrueba`, ya que no tiene utilidad. Véase el método `lecturaFicheroOK_casoPrueba_1` del código de la clase generada anterior.

Para el tipo de resultado **ERROR** en el elemento `resultado` se deberá colocar el nombre canónico de la excepción que se espera que se arroje al ejecutar el método a probar. Por ejemplo, en el caso del caso de prueba del método `lecturaFicheroError`, un test con tipo de resultado **ERROR**, en `resultado` se establece “`java.io.FileNotFoundException`”, que es el nombre canónico de la excepción que se espera que el método arroje al tratar de leer un fichero en una ruta que no existe (este test sirve por tanto para probar que el método exhibe un comportamiento deseable ante un escenario de error concreto). Véase el método `lecturaFicheroError_casoPrueba_1`.

El tipo de resultado **VARIABLE** es el más sencillo, ya que para él el código del método de prueba generado compara el resultado de la ejecución del método a probar directamente con el valor dado para el elemento resultado del tipo anotación **@AutoTestCasoPrueba**. Véase por ejemplo el cuerpo del siguiente método de prueba generado para tipo resultado **VARIABLE**:

```
@AutoTestGenerado
public boolean sumar_casoPrueba_1() {
    try {
        // instancia del objeto a probar
        AutoTestClaseAnotadaSumarRestar objTest = new AutoTestClaseAnotadaSumarRestar();

        // llamada con los argumentos especificados y recolección del resultado
        int resultado = objTest.sumar(0, 1);

        // presentación del resultado por consola
        System.out.println("AutoTest: sumar_casoPrueba_1: resultado = " + resultado);

        // comprobación de resultado: AUTO-TEST OK si el resultado es igual al esperado
        return resultado == 1;
    } catch (Throwable t) {
        // AUTO-TEST NOOK

        // presentación del error por consola
        System.out.println("AutoTest: sumar_casoPrueba_1: error: " + t);

        return false;
    }
}
```

Como hemos establecido en los requisitos de procesamiento para el tipo anotación **@AutoTest**, los métodos generados a partir de los casos de prueba dados por los **@AutoTestCasoPrueba**, están anotados con un tipo anotación **@AutoTestGenerado** con retención **RUNTIME**.

@AutoTestGenerado no es un tipo anotación que vaya a ser procesado por ningún otro procesador, si no que se utilizará como anotación marcadora para realizar el descubrimiento de los métodos de prueba generados en tiempo de ejecución mediante la clase **AutoTestRunner** y, de esta manera, poder ejecutarlos y obtener sus resultados. Véase la página siguiente.

Ejecutando el comando que invoca al **AutoTestRunner** (la segunda de las invocaciones dadas para este ejemplo, que invoca a **java** en lugar de a **javac**) obtenemos la ejecución de los métodos de prueba descubiertos y sus correspondientes resultados.

Al final se muestran unos totales que informan del total de clases, tests ejecutados, test correctos OK y tests incorrectos NOOK (nótese que, aunque se ven errores durante la ejecución de los métodos de prueba descubiertos de los “AutoTest”, el resultado de dichos “AutoTest” es OK porque precisamente lo que se quiere con dichos tests es que los métodos originales fallen).

Lo siguiente es la salida por pantalla obtenida al ejecutar la aplicación **AutoTestRunner**:

```
AutoTest runner: iniciando...
AutoTest runner: descubrimiento: paquete raiz: anotaciones.procesamiento
AutoTest runner: descubrimiento: clases AutoTest: 3
AutoTest runner: tests: iniciando ejecucion de tests...
AutoTest runner: tests: clase 1/3:
anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaLecturaFicherosAutoTest
AutoTest: lecturaFicheroOK_casoPrueba_1: metodo void ejecutado sin errores.
AutoTest runner: tests: test lecturaFicheroOK_casoPrueba_1: OK
AutoTest: lecturaFicheroError_casoPrueba_1: error: java.io.FileNotFoundException:
C:\ruta\fichero\fichero_no_existente.txt
AutoTest runner: tests: test lecturaFicheroError_casoPrueba_1: OK
AutoTest runner: tests: clase 2/3:
anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaMultiplicarDividirAutoTest
AutoTest: multiplicar_casoPrueba_1: resultado = 1
AutoTest runner: tests: test multiplicar_casoPrueba_1: OK
AutoTest: multiplicar_casoPrueba_2: resultado = 2
AutoTest runner: tests: test multiplicar_casoPrueba_2: OK
AutoTest: multiplicar_casoPrueba_3: resultado = 3
AutoTest runner: tests: test multiplicar_casoPrueba_3: OK
AutoTest: dividir_casoPrueba_1: resultado = 1
AutoTest runner: tests: test dividir_casoPrueba_1: OK
AutoTest: dividir_casoPrueba_2: resultado = 2
AutoTest runner: tests: test dividir_casoPrueba_2: OK
AutoTest: dividir_casoPrueba_3: resultado = 3
AutoTest runner: tests: test dividir_casoPrueba_3: OK
AutoTest: dividir_casoPrueba_4: error: java.lang.ArithmetricException: / by zero
AutoTest runner: tests: test dividir_casoPrueba_4: OK
AutoTest runner: tests: clase 3/3:
anotaciones.procesamiento.java67.ejemplos.autoTestAutoTest.AutoTestClaseAnotadaSumarRestarAutoTest
AutoTest: sumar_casoPrueba_1: resultado = 1
AutoTest runner: tests: test sumar_casoPrueba_1: OK
AutoTest: sumar_casoPrueba_2: resultado = 2
AutoTest runner: tests: test sumar_casoPrueba_2: OK
AutoTest: sumar_casoPrueba_3: resultado = 3
AutoTest runner: tests: test sumar_casoPrueba_3: OK
AutoTest: restar_casoPrueba_1: resultado = 0
AutoTest runner: tests: test restar_casoPrueba_1: OK
AutoTest: restar_casoPrueba_2: resultado = 1
AutoTest runner: tests: test restar_casoPrueba_2: OK
AutoTest: restar_casoPrueba_3: resultado = 2
AutoTest runner: tests: test restar_casoPrueba_3: OK
AutoTest runner: tests: ejecucion de tests finalizada.
AutoTest runner: tests: resultados: CLASES: 3
AutoTest runner: tests: resultados: TESTS: 15
AutoTest runner: tests: resultados: OK: 15
AutoTest runner: tests: resultados: NOOK: 0
AutoTest runner: finalizado.
```

Podemos hacer que falle cualquier test alterando los resultados de los casos de prueba a valores absurdos. Por ejemplo, si en la clase AutoTestClaseAnotadaSumarRestar cambiamos el resultado de cualquier caso de prueba a 666, el test correspondiente fallará al calcularse otro resultado:

```
@AutoTest(casosPrueba={
    @AutoTestCasoPrueba(argumentos="0|1", resultado="1", tipoResultado=AutoTestTipoResultado.VARIABLE),
    @AutoTestCasoPrueba(argumentos="1|1", resultado="2", tipoResultado=AutoTestTipoResultado.VARIABLE),
    @AutoTestCasoPrueba(argumentos="1|2", resultado="666", tipoResultado=AutoTestTipoResultado.VARIABLE)
})
public int sumar(int arg1, int arg2) {
    return arg1 + arg2;
}
```

El tercer caso de prueba del método sumar fallará porque $1+2$ no suma 666. Si lanzamos todo el proceso de nuevo, obtenemos el siguiente resultado para el tercer caso de prueba de sumar:

```
AutoTest: sumar_casoPrueba_3: resultado = 3
AutoTest runner: tests: test sumar_casoPrueba_3: NOOK
```

Finalmente, comentar que el cuerpo de AutoTestProcessor es similar al del tipo **@Proxy**, salvo por que se hace una “primera pasada” para extraer la lista de clases anotadas con al menos una anotación **@AutoTest**. Nótese el uso de un **TreeMap** (ordenado) con el nombre canónico de las clases como clave, para luego poder recorrer la lista de clases anotadas en orden lexicográfico ascendente por nombre canónico, como piden los requisitos del tipo anotación **@AutoTest**.

```
// PASADA INICIAL DE BÚSQUEDA DE CLASES CON MÉTODOS ANOTADOS

// realizamos una pasada por los elementos anotados para guardar una
// lista ORDENADA de las clases que contienen al menos un método anotado
TreeMap<String, TypeElement> elemsClasesConMetodosAnotados = new TreeMap<String, TypeElement>();

for (Element elemAnotado : elementosAnotados) {

    // PROCESAMIENTO DE ELEMENTOS ANOTADOS SUELTOS DESORDENADOS

    // sabemos que el elemento es un elemento de método -> down-cast
    ExecutableElement metodoElemAnotado = (ExecutableElement) elemAnotado;

    // COMPROBACIÓN: TIPO DECLARANTE NO ES UNA CLASE

    // obtenemos el tipo que declara este método (que tiene que ser exclusivamente una clase)
    Element elemTipoDeclaranteMetodo = metodoElemAnotado.getEnclosingElement();

    // ¿el tipo donde de está declarado el método anotado NO es una clase?
    if (elemTipoDeclaranteMetodo.getKind() != ElementKind.CLASS) {

        // la anotación @AutoTest se ha puesto sobre un método declarado
        // dentro de un tipo que NO es una clase -> ERROR
        this.messager.printMessage(Kind.ERROR,
            "@AutoTest: error: @AutoTest sólo es aplicable "
            + "a métodos de clases (" + elemTipoDeclaranteMetodo + " no es una clase)",
            elemAnotado);
        // y pasamos a procesar el siguiente elemento anotado
        continue;
    }

    // efectivamente la anotación está hecha sobre una clase -> down-cast
    TypeElement claseDeclaranteElem = (TypeElement) elemTipoDeclaranteMetodo;

    // DECLARACIÓN OK -> GUARDAMOS LA CLASE DONDE SE DECLARA LA ANOTACIÓN

    // añadimos dicha clase al map ordenado de clases con métodos anotados
    elemsClasesConMetodosAnotados.put(
        claseDeclaranteElem.getQualifiedName().toString(), claseDeclaranteElem);
}
```

El esquema del cuerpo principal de procesamiento del AutoTestProcessor es el siguiente:

```
// PASADA DE PROCESAMIENTO CLASE POR CLASE EN ORDEN

// una vez tenemos descubiertas las clases anotadas con al menos un método @AutoTest,
// vamos procesando dichas clases en el orden lexicografico dado por su nombre
// cualificado, que era la clave del map ordenado
Collection<TypeElement> colecciónClasesAProcesar = elemsClasesConMetodosAnotados.values();

// paquete del procesador de anotaciones (para los imports de las nuevas clases generadas)
String paqueteProcesador = this.getClass().getPackage().getName();

// procesamos una a una y en el orden dado por la colección ordenada
// todos los elementos de las clases anotadas con métodos de prueba
for (TypeElement claseAProcesarElem : colecciónClasesAProcesar) {

    // PREPARACIÓN DEL FICHERO DE LA NUEVA SUBCLASE

    this.messager.printMessage(Kind.NOTE,
        "procesador: procesamiento: clase: " + claseAProcesarElem.getQualifiedName());

    // propiedades de la clase original
    PackageElement pqElem = (PackageElement) claseAProcesarElem.getEnclosingElement();
    String paqueteClaseOriginal = pqElem.getQualifiedName().toString();
    String nombreClaseOriginal = claseAProcesarElem.getSimpleName().toString();
    String nombreCanonicoClaseOriginal = paqueteClaseOriginal + "." + nombreClaseOriginal;

    // propiedades de la nueva clase a generar
    String paqueteNuevaClase = pqElem.getQualifiedName().toString() + "AutoTest";
    String nombreNuevaClase = claseAProcesarElem.getSimpleName().toString() + "AutoTest";
    String nombreCanonicoNuevaClase = paqueteNuevaClase + "." + nombreNuevaClase;

    // GENERACIÓN DEL NUEVO FICHERO

    this.messager.printMessage(Kind.NOTE,
        "procesador: procesamiento: fichero: generando fichero...");

    // creamos el manejador del nuevo fichero...
    JavaFileObject fichero = null;
    PrintWriter ficheroWriter = null;

    try {

        // CREACIÓN DEL FICHERO

        // creamos un manejador de fichero asociado al nombre canónico de la nueva clase
        this.messager.printMessage(Kind.NOTE,
            "procesador: procesamiento: fichero: creando manejador fichero...");
        fichero = this.filer.createSourceFile(nombreCanonicoNuevaClase);
        this.messager.printMessage(Kind.NOTE,
            "procesador: procesamiento: fichero: manejador de fichero creado.");

        // creamos el flujo escritor del nuevo fichero
        this.messager.printMessage(Kind.NOTE,
            "procesador: procesamiento: fichero: abriendo fichero...");
        ficheroWriter = new PrintWriter(fichero.openWriter());
        this.messager.printMessage(Kind.NOTE,
            "procesador: procesamiento: fichero: fichero abierto.");

        // CÓDIGO FUENTE DE LA NUEVA SUBCLASE

        // ... generación del código de la nueva subclase con los métodos de prueba ...

    } catch (IOException e) { // ...
    } finally { // ...
    }
}
```

El bloque de **generación del código fuente de la nueva subclase** es el siguiente:

```
// CÓDIGO FUENTE DE LA NUEVA SUBCLASE

this.messenger.printMessage(Kind.NOTE,
    "procesador: procesamiento: fichero: escribiendo fichero...");

// paquete
ficheroWriter.println("package " + paqueteNuevaClase + ";" + "\n");

// imports

// import previo: para la anotación @Generated
ficheroWriter.println("import javax.annotation.Generated;" + "\n");

// import del paquete del procesador, así como el paquete de la clase original
// (si son diferentes, ya que en el caso de nuestro ejemplo coinciden)
if (!paqueteProcesador.equals(paqueteClaseOriginal)) {
    ficheroWriter.println("import " + paqueteProcesador + ".*;");
}
ficheroWriter.println("import " + paqueteClaseOriginal + ".*;");
ficheroWriter.println("");

// cabecera javadoc de la nueva clase
ficheroWriter.println("/**");
ficheroWriter.println("* Clase de auto-tests generada a partir de los casos de prueba de la");
ficheroWriter.println("siguiente clase original:");
ficheroWriter.println("* " + nombreCanonicoClaseOriginal + ".");
ficheroWriter.println("*/");

// anotación @Generated (nueva de Java SE 6) sobre la definición de la clase;
// valores de los elementos (establecidos según las convenciones de la
// documentación oficial de Java SE 6+ sobre el tipo anotación @Generated):
// value: nombre canónico de la clase generadora de código
// comments: nombre de la utilidad de generación de código o cualquier otro comentario libre
// date: fecha de la generación de código en formato ISO 8601
ficheroWriter.println("@Generated(\""
    + "value = { \\" + this.getClass().getCanonicalName() + "\\}, " + "\\n"
    + "comments = \\" + "Procesador de anotaciones ejemplo @AutoTest." + "\\", "
    + "date = \\" + formatoFechaISO8601.format(new Date()) + "\\")");

// definición de la nueva clase
ficheroWriter.println("public class " + nombreNuevaClase + " {}");

// MÉTODOS DE PRUEBA

// bloque de métodos de prueba
ficheroWriter.println("    ");
ficheroWriter.println("    // =====");
ficheroWriter.println("    // MÉTODOS DE PRUEBA");
ficheroWriter.println("    // =====");
ficheroWriter.println("    ");

// recuperamos los métodos de la clase de prueba actual
Collection<ExecutableElement> metodosClaseAProcesar =
    ElementFilter.methodsIn(claseAProcesarElem.getEnclosedElements());
```

```

// añadimos un método de prueba por cada caso de prueba de cada método anotado
for (ExecutableElement metodoElem : metodosClaseAProcesar) {

    // recuperamos la anotación @AutoTest del elemento del método
    AutoTest anotacionAutoTest = metodoElem.getAnnotation(AutoTest.class);

    // si el método no está anotado con @AutoTest -> pasamos al siguiente
    if (anotacionAutoTest == null) continue;

    // obtenemos el array de casos de prueba
    // para los que hay crear los métodos de prueba
    AutoTestCasoPrueba[] casosPrueba = anotacionAutoTest.casosPrueba();

    // COMPROBACIÓN: DECLARADO AL MENOS 1 CASO DE PRUEBA

    // si el array está vacío -> no hay ningún caso de prueba -> ERROR
    if (casosPrueba.length == 0) {

        // arrojamos un error a la salida del compilador
        this.messager.printMessage(Kind.ERROR,
            "@AutoTest: error: @AutoTest requiere de al menos un caso "
            + "de prueba definido con una anotación @AutoTestCasoPrueba",
            metodoElem);

        // pasamos a procesar el siguiente método
        continue;
    }

    // GENERACIÓN DE MÉTODOS DE PRUEBA

    // método con al menos un caso de prueba -> generamos sus métodos de prueba
    for (int i=1; i<=casosPrueba.length; i++) {

        // referencia al caso de prueba actual
        AutoTestCasoPrueba casoPrueba = casosPrueba[i-1];

        // nombre del método de prueba:
        // nombre del método anotado + "_casoPrueba_" + i
        String nombreMetodoPrueba =
            metodoElem.getSimpleName() + "_casoPrueba_" + i;

        this.messager.printMessage(Kind.NOTE,
            "procesador: procesamiento: "
            + "generando método de prueba: " + nombreMetodoPrueba);

        // generamos el método de prueba
        this.generarMetodoPrueba(metodoElem, casoPrueba, nombreMetodoPrueba, ficheroWriter);
    }
}

// for (ExecutableElement metodoElem : metodosClaseAProcesar)

// cerramos el cuerpo de la clase
ficheroWriter.println("}");

```

Como puede deducirse del código anterior, se escribe al flujo de salida del nuevo fichero de código generado (**ficheroWriter**) una parte inicial del código de la clase que se podría llamar “estático” o “fijo”: paquete, imports, javadoc y definición de la clase, y finalmente el comentario que introduce el bloque **MÉTODOS DE PRUEBA**. Y luego, para la parte que requiere de código “dinámico” o “variable” se monta un **for** que iterará sobre los métodos de la clase anotada descubierta a procesar. Se comprueba si el método está realmente anotado y tiene al menos un caso de prueba (requisitos mínimos exigibles para poder generar al menos un método de prueba a partir de la información dada por la anotación). Si todo está correcto, se llama al método auxiliar **generarMetodoPrueba**, que será el encargado de generar finalmente el código concreto del método de prueba en función de los valores de la anotación **@AutoTestCasoPrueba** del método anotado. El código de dicho método se muestra a continuación.

Debido a la mayor complejidad de la generación de código necesaria para crear los métodos de prueba de las “clases AutoTest”, se ha refactorizado la generación del código fuente al método generarMetodoPrueba, que genera 3 tipos de métodos de prueba, con ligeras diferencias entre ellos, en función de que el tipo de resultado sea **VARIABLE**, **VOID** o **ERROR**.

Como ya hemos mencionado, en caso de que el resultado sea **VARIABLE** habrá que recoger el valor del elemento resultado de la anotación y compararlo con el resultado obtenido de la invocación del método original con sus correspondientes argumentos. En caso de ser **VOID**, no tiene sentido recoger ningún resultado y simplemente se retorna **true** si no ha habido ninguna excepción y, finalmente, en caso de ser un resultado de tipo **ERROR**, hay que comparar el nombre canónico de la clase excepción capturada con la esperada, dado por el elemento resultado.

Lo siguiente es el cuerpo del método generarMetodoPrueba con la sangría suprimida en las líneas de escritura para tratar de que desborden la anchura de la página lo menos posible:

```
private void generarMetodoPrueba(ExecutableElement metodoElem, AutoTestCasoPrueba casoPrueba,
    String metodoPrueba, PrintWriter p) {

    // propiedades útiles para la generación del cuerpo del método
    String metodo = metodoElem.getSimpleName().toString();
    TypeElement claseDeclaranteMetodo = (TypeElement) metodoElem.getEnclosingElement();
    String claseObjAProbar = claseDeclaranteMetodo.getSimpleName().toString();
    String claseResultado = metodoElem.getReturnType().toString();
    String[] argumentosArray = casoPrueba.argumentos().split("\\\\|");
    String argumentos = Arrays.toString(argumentosArray).replace("[", "").replace("]", "");
    String resultado = casoPrueba.resultado();

    // GENERACIÓN DEL CUERPO DEL MÉTODO EN FUNCIÓN DEL TIPO DE RESULTADO ESPERADO

    if (casoPrueba.tipoResultado() == AutoTestTipoResultado.VARIABLE) {

        // TIPO DE RESULTADO: VARIABLE

        p.println("@AutoTestGenerado");
        p.println("public boolean " + nombreMetodoPrueba + "() {" +
            "try {" +
                " // instancia del objeto a probar" +
                " " + claseObjAProbar + " objTest = new " + claseObjAProbar + "();" +
                " // llamada con los argumentos especificados y recolección del" +
                " " + claseResultado + " resultado = objTest." + metodo + "(" + argumentos + ");";
            " " +
            " // presentación del resultado por consola" +
            " System.out.println(\"AutoTest: " + metodoPrueba + ": resultado = \" +" +
            resultado + ");" +
            " // comprobación de resultado: AUTO-TEST OK si el resultado es igual al esperado" +
            " return resultado == " + resultado + ";" +
            " } catch (Throwable t) {" +
            " " +
            " // AUTO-TEST NOOK" +
            " " +
            " // presentación del error por consola" +
            " System.out.println(\"AutoTest: " + metodoPrueba + ": error: \" + t);";
            " " +
            " return false;" +
            " }");
        p.println("}");
        p.println("");
    }
}
```

```

} else if (casoPrueba.tipoResultado() == AutoTestTipoResultado.VOID) {

    // TIPO DE RESULTADO: VOID

p.println("@AutoTestGenerado");
p.println("public boolean " + metodoPrueba + "() {" +
p.println("    ");
p.println("        try {" +
p.println("            ");
p.println("                // instancia del objeto a probar");
p.println("                " + claseObjAProbar + " objTest = new " + claseObjAProbar + "();");
p.println("            ");
p.println("                // llamada con los argumentos especificados");
p.println("                objTest." + metodo + "(" + argumentos + ");");
p.println("            ");
p.println("                // presentación del resultado por consola");
p.println("                System.out.println(\"AutoTest: " + metodoPrueba + ": metodo void");
ejecutado sin errores.\");");
p.println("        ");
p.println("        // comprobación de resultado: AUTO-TEST OK si no se arroja ninguna excepción");
p.println("        return true;");
p.println("    ");
p.println("} catch (Throwable t) {" +
p.println("    ");
p.println("        // AUTO-TEST NOOK");
p.println("        ");
p.println("        // presentación del error por consola");
p.println("        System.out.println(\"AutoTest: " + metodoPrueba + ": error: \" + t);");
p.println("        ");
p.println("        return false;");
p.println("    ");
p.println("} ");
p.println("} ");
p.println("} ");

} else if (casoPrueba.tipoResultado() == AutoTestTipoResultado.ERROR) {

    // TIPO DE RESULTADO: ERROR

p.println("@AutoTestGenerado");
p.println("public boolean " + metodoPrueba + "() {" +
p.println("    ");
p.println("        try {" +
p.println("            ");
p.println("                // instancia del objeto a probar");
p.println("                " + claseObjAProbar + " objTest = new " + claseObjAProbar + "();");
p.println("            ");
p.println("                // llamada con los argumentos especificados y recolección del");
p.println("                // resultado");
p.println("                objTest." + metodo + "(" + argumentos + ");");
p.println("            ");
p.println("                // comprobación de resultado: AUTO-TEST NOOK si no hay errores");
p.println("                return false;");
p.println("            ");
p.println("} catch (Throwable t) {" +
p.println("    ");
p.println("        // presentación del error por consola");
p.println("        System.out.println(\"AutoTest: " + metodoPrueba + ": error: \" + t);");
p.println("        ");
p.println("        // comprobación de resultado: AUTO-TEST OK si se recibe una excepción del");
p.println("        // tipo esperado en el resultado");
p.println("        return t.getClass().getCanonicalName().equals(\"" + resultado + "\");");
p.println("    ");
p.println("} ");
p.println("} ");

} // if

} // generarMetodoPrueba

```

15.6.- Depuración de procesadores de anotaciones.

La depuración de procesadores de anotaciones presenta una serie de problemas específicos, debido a que en el caso general el procesamiento se lleva a cabo antes incluso de que la compilación se haya llevado a cabo.

Así las cosas, los procesadores de anotaciones JSR 269 dan bastantes problemas a la hora de ser depurados, ya que, aunque se coloquen puntos de ruptura (o “breakpoints”) en ciertas líneas de código, en la mayoría de entornos la ejecución no se detendrá en ellos.

Para obtener información acerca de lo que ocurre en el entorno de ejecución de los procesadores de anotaciones, se utilizan los métodos `printMessage` de una instancia de la interfaz `Messager`, instanciada a partir del método `ProcessingEnvironment.getMessager()`, tal y como se ve en el siguiente ejemplo:

```
@Override
public boolean process(Set<? extends TypeElement> tiposAnotacion, RoundEnvironment
roundEnv) {

    msg.printMessage(Kind.NOTE, "process: iniciando ronda de procesamiento...");
    msg.printMessage(Kind.NOTE, "process: tiposAnotacion = " + tiposAnotacion);
    msg.printMessage(Kind.NOTE, "process: roundEnv = " + roundEnv);
    msg.printMessage(Kind.NOTE, "process: ronda final = " + roundEnv.processingOver());

    ...resto del código ...
}
```

Mediante los métodos `printMessage` podemos imprimir cualquier tipo de información por la salida del compilador, e incluso emitir warnings o errores de compilación si como su primer argumento pasamos `Kind.WARNING` o `Kind.ERROR` respectivamente.

Existe alguna alternativa que puede optimizar el proceso de depurar procesadores de anotaciones, como es la posibilidad de usar un plugin específico del entorno Eclipse para esta tarea. Los detalles del proceso quedan fuera del alcance de este manual, ya que son una alternativa limitada y que carece de soporte oficial, por lo que no es posible confiar en su correcto funcionamiento en todas las versiones del entorno. No obstante, por el interés intrínseco que tiene de cara a mejorar la eficiencia del proceso de depuración de un procesador de anotaciones en la práctica, a continuación se dejan aquí las referencias a un vídeo y un tutorial que muestran cómo configurar Eclipse para lograr esto:

Java annotation processing in Eclipse

<https://www.youtube.com/watch?v=PjUaHkUsgzo>

Debugging annotation processor (Eclipse based)

https://code.google.com/archive/p/acris/wikis/AnnotationProcessing_DebuggingEclipse.wiki

16.- TRANSFORMACIÓN DE CÓDIGO FUENTE.

16.1.- Introducción.

A lo largo de este manual ya hemos examinado las APIs de procesamiento de anotaciones así como las distintas APIs auxiliares que nos ayudaban en dicho propósito. Como hemos visto, el procesamiento de anotaciones es una funcionalidad del lenguaje Java que puede ser utilizada en múltiples escenarios y tiene un enorme potencial.

Sin embargo, las APIs de procesamiento de anotaciones ofrecidas por las diversas especificaciones Java oficiales tienen una importante limitación: **un procesador de anotaciones no puede modificar el código de una clase existente.**

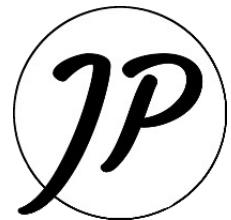
Es decir, que usando el procesamiento de anotaciones estándar sólo se pueden crear nuevas clases, pero nunca modificar las clases ya existentes. Esto no supone una restricción grave en la mayoría de escenarios de uso del procesamiento de anotaciones, ya que los procesadores típicamente generan nuevos ficheros (ya sean .java, .class u otro tipo de ficheros) como resultado de su procesamiento. Siendo así, la modificación del código fuente de los ficheros Java procesados se consideró una funcionalidad innecesaria e incluso perniciosa, debido a los problemas de diseño que podría entrañar.

No obstante, en algunos escenarios de uso más raros pueden darse casos en los que la funcionalidad de transformación de código fuente Java mediante procesamiento de anotaciones sea deseable, por lo que vamos a dedicar este apartado a enumerar algunas librerías Java que realizan esta funcionalidad en mayor o menor medida. De esta forma, el lector interesado podrá ahondar en cada una de ellas y elegir si utilizarla para implementarla en su propio software.

Además de las librerías presentadas en este apartado, que fundamentalmente sirven para modificar los árboles sintácticos de las clases Java que parsean, existen otras APIs que permiten manipular el *bytecode* de los ficheros de clase. Estas librerías no tienen tanta expresividad como las que modifican los árboles sintácticos de código, pero también pueden ser útiles en casos en los que se necesiten sólo transformaciones sencillas de las clases originales. Estas librerías de manipulación de *bytecode* están por lo general mucho más maduras, más documentadas y más ampliamente difundidas que las que manipulan el código fuente. Las librerías de manipulación de bytecode Java más populares son **Javassist** [<http://www.javassist.org>] (vista en este manual), **cglib** [<https://github.com/cglib/cglib>] y **ASM** [<http://asm.ow2.org>].

16.1.- Javaparser.

Javaparser [<http://javaparser.org>], según la definen sus autores, es una librería que permite parsear, analizar, transformar y generar código Java. En este caso en particular nos interesa la funcionalidad de transformación que, como hemos dicho, es la única que las APIs oficiales no ofrecen.



Javaparser no dispone actualmente de una buena documentación y requerirá de un esfuerzo de aprendizaje e investigación para familiarizarse con su API y ser capaz de implementar el código de transformación de clases necesario. Para aplacar este problema, se está escribiendo un libro llamado «Javaparser: Visited» [<https://leanpub.com/javaparservisited>] por parte de los autores de la librería.

Javaparser se reseña en primer lugar en este apartado porque es una API relativamente madura y que ha sido adoptada por un buen número de proyectos externos que ya hace tiempo que la están usando,

lo cual la convierte en principio en la opción más interesante de partida, especialmente en el caso de que el libro mencionado solucione en buena parte el problema de su pobre documentación.

Por supuesto, el procesamiento de clases para su transformación no se integra ni forma parte del proceso de procesamiento de anotaciones estándar, si no que, en todo caso, habría de ejecutarse como parte de un proceso de build, ya fuera anterior o posterior a la compilación.

16.2.- Spoon.

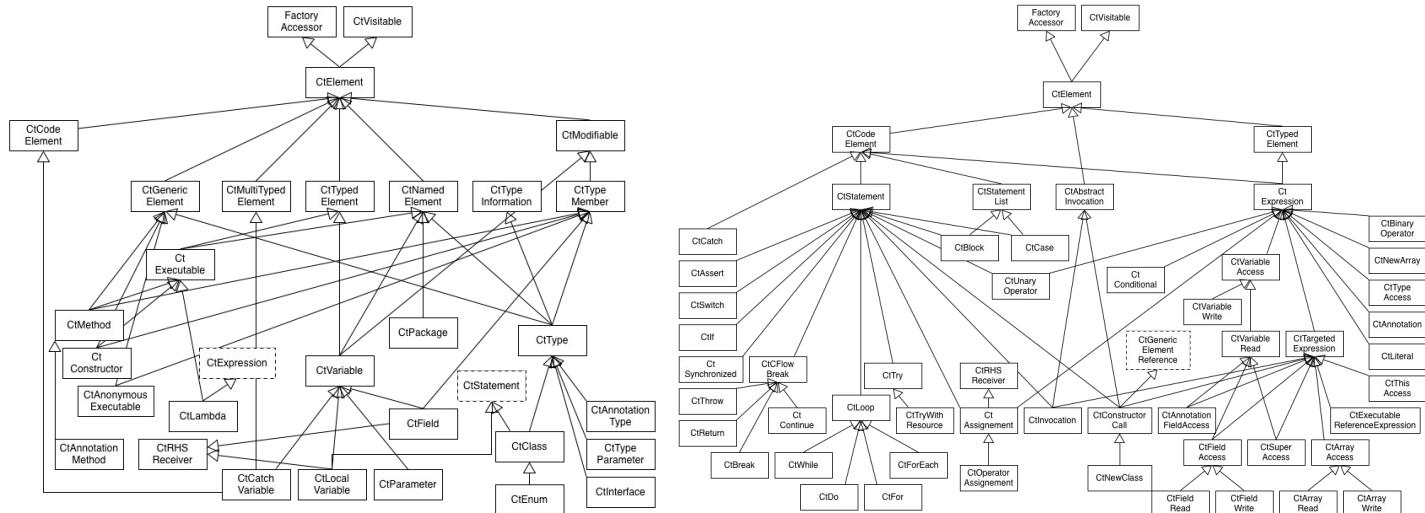
Spoon [<http://spoon.gforge.inria.fr>] es otra librería que permite el parseo, análisis y transformación de código fuente Java a través de la modificación de los árboles sintácticos. Esto se realiza a través del parseo de las clases a analizar y su representación en su metamodelo correspondiente.

Spoon tiene una documentación más completa que Javaparser, así como múltiples ejemplos de código. Sin embargo, es posible que sea superado en ese aspecto por Javaparser cuando este publique su libro. Por tanto, en el aspecto de documentación tampoco destaca especialmente sobre el resto de librerías, ya que aunque sus aspectos básicos sí están bien explicados, nos encontramos con la misma carencia de documentación en su javadoc que en la mayoría de librerías de este tipo.

Según los creadores de Spoon, su metamodelo es fácil de comprender para los desarrolladores Java, aunque tiene un enorme número de clases que responden a 3 categorías distintas:

- 1) Elementos estructurales: clases, interfaces, variables, métodos, anotaciones y tipos enumerados.
 - 2) Código: modela los elementos sintácticos del código fuente como bloques o sentencias.
 - 3) Referencias: modela las referencias a los elementos de programa como referencias a tipos, etc.

A continuación siguen los diagramas de los elementos estructurales y de código de la documentación oficial de la librería en la que podemos ver el gran número de clases integradas en su metamodelo:



El diseño de Spoon está claramente influenciado tanto por las APIs oficiales de procesamiento de anotaciones como por Javassist. Spoon usa el concepto de “procesador” y, más específicamente, también de “procesador de anotaciones”. No obstante, a pesar del nombre tan parecido de sus clases de procesadores con las de las APIs oficiales, Spoon no es compatible con el procesamiento de anotaciones estándar, si no que usa clases que invocan el recorrido de los árboles sintácticos de las clases a procesar mediante el uso extensivo del patrón Visitor, de nuevo de una manera muy similar a las APIs oficiales. A pesar de esto, la implementación de la transformación de código fuente con Spoon requerirá del implementador también un largo proceso de aprendizaje e investigación para dominar adecuadamente su API.

16.3.- Project Lombok.

Project Lombok [<https://projectlombok.org>] es una librería cuyo objetivo es eliminar la necesidad de código superfluo en las clases Java, como pueden ser los métodos getter/setter, hashCode, equals o toString. Esto lo consigue con anotaciones predefinidas que, gracias al procesamiento de anotaciones, generan todo ese código superfluo en tiempo de compilación.

Project Lombok no ofrece funcionalidades de transformación de código fuente de manera directa pero las hace para los objetivos que persigue mediante transformaciones sobre los árboles sintácticos de código fuente Java, pero a diferencia de Javaparser o Spoon, **Project Lombok transforma el código de las clases Java usando la implementación oficial de las APIs de procesamiento de anotaciones.**



Esto aparentemente contradice lo que habíamos dicho de que las APIs oficiales de procesamiento de anotaciones no podían modificar el código fuente de las clases Java. ¿Cómo es entonces posible que Project Lombok logre hacerlo? La respuesta resulta realmente interesante y es una opción a valorar para implementar la transformación de código fuente a través de clases de la implementación oficial, con la ventaja principal que ello implica de no tener que aprender a usar las APIs de otras librerías.

Y es que **Project Lombok realiza procesamiento de anotaciones con transformación de código fuente Java con usando las clases de la implementación oficial de Oracle porque emplea código hackeado que usa funcionalidades no públicas.**

En el caso de la implementación de Oracle, se usa la clase privada **JavacAnnotationProcessor** como procesador de anotaciones y sus respectivas clases privadas de **RoundEnvironment** y **TypeElement**, que tienen funciones para modificar los árboles sintácticos de las clases Java. Además de esto, Project Lombok usa otro hackeo para lograr lo mismo con la API del compilador de Eclipse.

El beneficio fundamental de estos hackeos es que dichas modificaciones sintácticas son visibles para el compilador, consiguiéndose el efecto deseado de poder modificar el código de clases Java en tiempo de compilación. Lombok los emplea para generar código superfluo usando su anotación **@Data**. En la imagen se ve cómo se usa sobre la clase Mountain. Nótese todos los métodos que no aparecen en el código original y sí lo hacen en el Outline de la clase, ya que han sido generados y son visibles para el compilador y el entorno de Eclipse.

The screenshot shows the Eclipse IDE interface. On the left is the code editor with the file "Mountain.java" open, containing the following code:

```
1 import lombok.Data;
2
3
4 @Data
5 public class Mountain {
6     private final String name;
7     private final double latitude, longitude;
8     private String country;
9 }
```

On the right is the "Outline" view, which lists all the generated methods and fields for the Mountain class. The methods listed are:

- import declarations
- Mountain
- c Mountain(String, double)
- getName(): String
- getLatitude(): double
- getLongitude(): double
- getCountry(): String
- setCountry(String) void
- toString(): String
- equals(Object): boolean
- hashCode(): int
- name: String
- latitude: double
- longitude: double
- country: String

No obstante, por estos mismos motivos, el uso de Project Lombok entraña también un importante riesgo: al usar APIs no públicas, estas podrían cambiar o desaparecer en futuras versiones de las APIs oficiales de Java o del compilador de Eclipse, lo cual rompería la funcionalidad de la librería.

Lo más importante que nos enseña Project Lombok es que podemos inspirarnos en su método para implementar nuestras propias transformaciones de código fuente Java en tiempo de compilación. Para ello, podemos consultar el código fuente de Lombok [<https://github.com/rzwitserloot/lombok>] para ver cómo funciona, o el código fuente de las clases de implementación de procesamiento de anotaciones de Oracle, como **JavacAnnotationProcessor**, para ver las funcionalidades que nos ofrecen exactamente y cómo manejarnos con ellas para lograr nuestro objetivo concreto.

17.- NOVEDADES EN JAVA SE 7.

Las novedades en la especificación del lenguaje Java (Java Language Specification o JLS) provocan lógicamente cambios en las APIs que lo modelan. Estos cambios, que son necesarios para acomodar dichas novedades, pueden afectar al procesamiento de anotaciones, ya que en él se manejan abstracciones que representan dichos elementos del lenguaje.

Así pues, dado que las APIs de procesamiento de anotaciones y sus APIs relacionadas (APIs de reflexión, Language Model API, API de procesamiento JSR 269, Java Compiler API y Compiler Tree API) deben acomodarse a los cambios e innovaciones en el lenguaje Java, le dedicaremos a continuación un apartado introductorio a explicar las novedades a nivel general introducidas en el lenguaje y la plataforma de desarrollo en Java SE 7. A continuación, en un segundo apartado, se tratan los cambios en aquellas APIs relacionadas con el procesamiento de anotaciones que se han visto afectadas (no todas tienen por qué verse afectadas).

17.1.- Novedades generales en Java SE 7.

El 28 de Julio de 2011, se liberaba **Java SE 7** (con el nombre clave o *codename* de “**Dolphin**”), desarrollada bajo la “especificación paraguas” [JSR 336](#), siguiendo el proceso típico establecido a través del JCP. Java SE 7 fue la primera versión de la plataforma Java que fue desarrollada de principio a fin en el seno de [OpenJDK](#), el portal de desarrollo creado por Sun para la gestión del desarrollo *open source* de Java. Y es que, desde Java SE 7, OpenJDK es la implementación de referencia (reference implementation o RI). Gracias a la savia nueva proporcionada por la comunidad *open source* y al nuevo impulso de Oracle, Java SE 7 sí que introdujo muchas pequeñas mejoras en la plataforma Java.

Las **mejoras más importantes introducidas en Java SE 7** son las siguientes:

- Pequeñas [mejoras al lenguaje Java](#) agrupadas bajo el nombre de [Project Coin \(JSR 334\)](#):
 - Sentencia `try` con manejo automático de recursos: [try-with-resources](#).
 - [Multi-catch](#) de varias excepciones e [inferencia de tipos de excepciones propagadas](#).
 - Inferencia de tipos en creación de instancias genéricas: [operador diamante <>](#).
 - Uso de [variables de clase String](#) en los `case` de la sentencia `switch`.
 - [@SafeVarargs](#) para indicar manipulación segura frente a [polución del heap](#) en varargs.
 - [Literales binarios](#) en tipos enteros y [guiones bajos en los literales de tipo numérico](#).
- [Soporte para lenguajes dinámicos](#) con `invokedynamic` en la Máquina Virtual Java ([JSR 292](#)).
- [Mejoras de concurrencia](#): API [fork/join](#) y nuevas clases [ThreadLocalRandom](#) y [Phaser](#).
- [NIO.2 \(JSR 203\)](#) E/S con más independencia de la plataforma, metadatos, enlaces simbólicos.
- Interfaces gráficas: [Mejoras en Swing: mezcla de componentes pesados y ligeros](#).
- Interfaces gráficas: [JLayer](#), [transparencias y formas no rectangulares](#), [Nimbus Look and Feel](#).
- Gráficos: [Mejoras en Java 2D](#): motor [XRender](#) para [X11 XRender](#) (desactivado por defecto).
- Comunicaciones: Mejor soporte de librerías para nuevos protocolos: [SCTP](#), [SDP](#) y [TLS 1.2](#).
- Seguridad: [Mejoras de Seguridad](#): Soporte de [criptografía de curva elíptica](#) y otras mejoras.
- Rendimiento: Ordenación de arrays mediante [Timsort](#) en lugar de [Merge sort](#).
- Rendimiento: [Punteros de 64 bits comprimidos](#) para ganar eficiencia en MV's \leq 4GB memoria.

La lista completa de mejoras introducidas en Java SE 7 puede encontrarse en la página web oficial de Java SE 7 del portal OpenJDK [<http://openjdk.java.net/projects/jdk7/features/>] y en OTN [<http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html>]

17.2.- Novedades Java SE 7 en APIs de procesamiento.

NOTA: A veces la novedad son los tipos (clases y/o interfaces) en sí mismos, otras veces lo es la inclusión de nuevos campos y/o métodos. Para diferenciarlo claramente, los elementos nuevos aparecerán en color verde en los siguientes apartados.

17.2.1.- Novedades Java SE 7: API de reflexión.

La API de reflexión (clases `java.lang.Class`, `java.lang.Package` y las clases del paquete `java.lang.reflect`) apenas introdujeron mejoras en Java SE 7. De hecho, la [documentación oficial sobre las mejoras de las APIs de reflexión](#) no refiere ninguna mejora.

No obstante, explorando la documentación de las APIs de reflexión de Java SE 7 sí que podemos encontrar algunas pocas, aunque no están relacionadas en nada con el procesamiento de anotaciones. En `java.lang.reflect.Modifier` se añadieron una serie de métodos estáticos para poder recuperar los modificadores aplicados sobre distintos elementos de código en forma de máscara OR de bits representado por un número entero: `classModifiers`, `interfaceModifiers`, `constructorModifiers`, y `fieldModifiers`.

17.2.2.- Novedades Java SE 7: Language Model API.

La llegada de Java SE 7 sí que introdujo algunas novedades en la Language Model API. Además de corregir algún bug, se introdujeron algunos elementos nuevos debidos a los cambios dados por el [Project Coin](#) y se hicieron algunos pequeños retoques al diseño de la API.

Nueva “clase” (“kind”) de elemento RESOURCE_VARIABLE y nuevo tipo UNION

Language Model API – Novedades Java SE 7	
Paquete javax.lang.model	
Tipo	Campo(s) / Método(s)
SourceVersion	RELEASE_7 [campo que modela la versión de Java SE 7]
Paquete javax.lang.model.element	
Tipo	Campo(s) / Método(s)
ElementKind	RESOURCE_VARIABLE [campo que modela las variables de recurso declaradas en las nuevas sentencias try-with-resources]
Parameterizable	List<? extends TypeParameterElement> getTypeParameters()
Qualifiednameable	Name getQualifiedName()
Paquete javax.lang.model.type	
Tipo	Campo(s) / Método(s)
TypeKind	UNION [campo que modela los “tipos unión” aparecidos debido a la unión de tipos excepción en las nuevas expresiones multi-catch : try ... catch (Ex1 Ex2 ... ExN e)]
UnionType	List<? extends TypeMirror> getAlternatives() [para el ejemplo de arriba devolvería la lista Ex1, ..., ExN]
TypeVisitor	R visitUnion(UnionType t, P p)

Novedades relacionadas con los Visitor de Elementos en Java SE 7

Los Visitor de Elementos no sufren demasiados cambios, ya que no se han introducido nuevos tipos de elementos, sólo el “subtipo” o “clase” (“kind”) de elemento dado por la constante `ElementKind.RESOURCE_VARIABLE` correspondiente a las variables de recurso, pero esta es una “clase” de variables, por lo que sigue siendo una “clase” concreta de `VariableElement`, y la interfaz `ElementVisitor` no cambia.

Sólo el Visitor especializado por “clase” (“kind”) de elemento, `ElementKindVisitor7` sí que añade el nuevo `visitVariableAsResourceVariable` para visitar las variables de recurso.

IMPORTANTE: El Visitor de Java SE 6 `ElementKindVisitor6` también añade el método `visitVariableAsResourceVariable`, pero, como es lógico, llama a `visitUnknown`. Aquí es donde se hace imprescindible que, si se implementó un `ElementKindVisitor6`, se hubiera redefinido el método `visitUnknown` para no ejecutar su implementación por defecto, que arroja una `UnknownElementException`. De no haberlo hecho, nuestro `ElementKindVisitor6` funcionaría correctamente en Java SE 6, pero NO en Java SE 7, ya que fallaría en el momento en que se visitara una variable de recurso, puesto que se llamaría a `visitUnknown` y este método arrojaría la mencionada excepción. En ese caso, itendríamos un procesador de anotaciones que únicamente sería válido para Java SE 6!, algo totalmente inaceptable.

Language Model API – Novedades Java SE 7 – Visitors de Elementos	
Paquete <code>javax.lang.model.util</code>	
Visitor	Método(s)
<code>AbstractElementVisitor7</code>	No define ningún método nuevo.
<code>SimpleElementVisitor7</code>	No define ningún método nuevo.
<code>ElementKindVisitor6</code>	<code>public R visitVariableAsResourceVariable(VariableElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>ElementKindVisitor7</code>	<code>public R visitVariableAsResourceVariable(VariableElement e, P p)</code> (que por defecto llamará a <code>defaultAction</code>)
<code>ElementScanner7</code>	No define ningún método nuevo.

Novedades de los Visitor de Valores de Anotaciones en Java SE 7

Los Visitor de Valores de anotaciones no sufren ninguna modificación de Java SE 6 a Java SE 7, ya que no se han definido nuevos tipos para los valores de los elementos de los tipos anotación. No obstante, a pesar de no ser necesarios, se han definido nuevas clases para Java SE 7. Como puede comprobarse viendo el código fuente (vía [DocJar](#), por ejemplo), las nuevas clases añadidas, `AbstractAnnotationValueVisitor7` y `SimpleAnnotationValueVisitor7`, se limitan a simplemente a heredar de sus versiones de Java SE 6 y definir los constructores necesarios. No siendo necesarios y para evitar confundir a los desarrolladores con más clases Visitor, hubiera sido más correcto no definir Visitors de Valores de Anotaciones específicos para Java SE 7.

Language Model API – Novedades Java SE 7 – Visitors de Valores de Anotaciones	
Paquete <code>javax.lang.model.util</code>	
Visitor	Método(s)
<code>AbstractAnnotationValueVisitor7</code>	No define ningún método nuevo.
<code>SimpleAnnotationValueVisitor7</code>	No define ningún método nuevo.

Novedades relacionadas con los Visitor de Tipos en Java SE 7

Los Visitor de Tipos reciben el nuevo “tipo unión”, que es el tipo contenedor que expresa la unión de tipos de excepciones en las expresiones multi-catch de las nuevas sentencias try-with-resources. Para ello, se introduce el nuevo `UnionType` y la constante `TypeKind.UNION`, por lo que la interfaz `TypeVisitor` también añade un método `visitUnion`, lo cual hace que todos los Visitor de tipos deban ahora implementar dicho método.

IMPORTANTE: Por supuesto, al incluir el nuevo `visitUnion` en `TypeVisitor`, los Visitor de tipos de Java SE 6 ahora también deben proporcionar una implementación de dicho método, pero, como es lógico, dicha implementación llamará a `visitUnknown`. Aquí es donde se hace imprescindible que, si se implementó un `TypeVisitor` para Java SE 6, se hubiera redefinido el método `visitUnknown` para no ejecutar su implementación por defecto, que arroja una `UnknownTypeException`. De no haberlo hecho, nuestro `TypeVisitor` funcionaría correctamente en Java SE 6, pero NO en Java SE 7, ya que fallaría en el momento en que se visitara un tipo unión, puesto que se llamaría a `visitUnknown` y este método arrojaría la mencionada excepción, en cuyo caso, intentaríamos un procesador de anotaciones válido únicamente para Java SE 6!, algo totalmente inaceptable.

Language Model API – Novedades Java SE 7 – Visitors de Elementos	
Paquete <code>javax.lang.model.util</code>	
Visitor	Método(s)
<code>AbstractTypeVisitor6</code>	<code>public R visitUnion(UnionType t, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>AbstractTypeVisitor7</code>	<code>public abstract R visitUnion(UnionType t, P p)</code>
<code>SimpleTypeVisitor6</code>	<code>public R visitUnion(UnionType t, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>SimpleTypeVisitor7</code>	<code>public R visitUnion(UnionType t, P p)</code> (que por defecto llamará a <code>defaultAction</code>)
<code>ElementKindVisitor6</code>	<code>public R visitUnion(UnionType t, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>ElementKindVisitor7</code>	<code>public R visitUnion(UnionType t, P p)</code> (que por defecto llamará a <code>defaultAction</code>)
<code>TypeKindVisitor6</code>	<code>public R visitUnion(UnionType t, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>TypeKindVisitor7</code>	<code>public R visitUnion(UnionType t, P p)</code> (que por defecto llamará a <code>defaultAction</code>)

NOTA: Nótese como `visitUnion` es `abstract` sólo para `AbstractTypeVisitor7` y, sin embargo, no lo es para `AbstractTypeVisitor6`. Esto es así porque la clase abstracta de Java SE 6 puede dar una implementación por defecto para `visitUnion`, que será llamar a `visitUnknown`, mientras que para el visitor de Java SE 7 el tipo unión no es un tipo nuevo ni distinto de cualquiera de los demás, por lo que no se puede fijar un comportamiento por defecto, si no que se deja la definición de dicha lógica al implementador. Los Visitor que heredan de `AbstractTypeVisitor7` ya sí que implementan el método siguiendo el contrato común de las demás implementaciones sencillas, que es llamar a `defaultAction`.

Nuevo: javax.lang.model.element: Interfaces Parameterizable y QualifiedNameable.

Como un pequeño retoque de diseño, se han añadido las interfaces autónomas (“[mixin](#)”) llamadas **Parameterizable** y **Qualifiednameable**. A efectos prácticos, esto no supone ningún cambio en ninguna de las clases de la Language Model API, ya que los métodos que definen dichas interfaces ya los implementaban en Java SE 6 las mismas clases implementadoras que ahora los implementan en Java SE 7.

Parameterizable define el método **getTypeParameters**, y de ella heredan las interfaces de tipos y métodos (**TypeElement** y **ExecutableElement**), que son los tipos de elementos del lenguaje Java que pueden definir parámetros de tipo en sus signaturas.

Qualifiednameable define el método **getQualifiedName**, para recuperar el nombre cualificado de un elemento. De esta interfaz heredan las interfaces de los paquetes y los tipos (**PackageElement** y **TypeElement**), que son los tipos de elementos con nombres cualificados.

Los métodos de **Parameterizable** y **Qualifiednameable** ya eran implementados por las interfaces mencionadas en Java SE 6, pero ahora dichas interfaces heredan de estas interfaces autónomas y, por tanto, sus instancias pueden asignarse a variables con el tipo de las interfaces, lo que permite agrupar los tipos de elementos en base a sus características, algo conveniente a la hora de crear condiciones lógicas sobre los elementos o signaturas de métodos más simples.

Ejemplo: Uso de **Parameterizable** en una condición lógica

Supongamos que queremos realizar algún tipo de procesamiento con los parámetros de tipo de un elemento. En cualquier momento podríamos hacer lo siguiente:

```
if (elemAnotado instanceof Parameterizable) {  
  
    // down-cast  
    Parameterizable elemParametrizable = (Parameterizable) elemAnotado;  
  
    // parámetros de tipo  
    List<? extends TypeParameterElement> parametrosTipo =  
        elemParametrizable.getTypeParameters();  
  
    // ... procesamiento de los parámetros de tipo ...  
}
```

Ejemplo: Uso de **Parameterizable** en la firma de un método

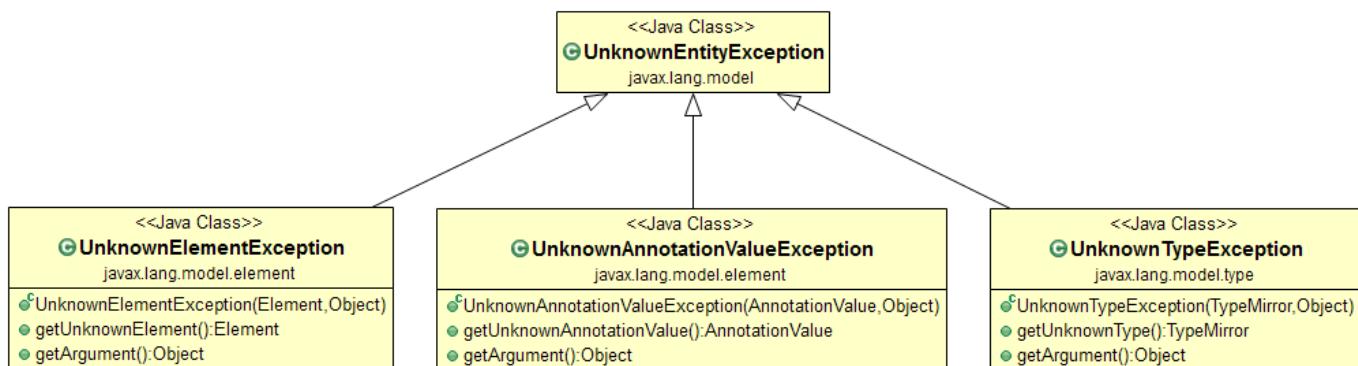
Supongamos que el procesamiento que queremos realizar con los parámetros de tipo de los elementos es muy complicado o engorroso, además de que nos resulta de uso frecuente. En ese caso, podríamos refactorizar dicho procesamiento en un método con un argumento de tipo **Parameterizable** gracias a la existencia de dicha nueva interfaz:

```
private void procesarParametrosTipo(Parameterizable elemParametrizable) {  
  
    // parámetros de tipo  
    List<? extends TypeParameterElement> parametrosTipo =  
        elemParametrizable.getTypeParameters();  
  
    // ... procesamiento de los parámetros de tipo ...  
}
```

Nuevo: Excep. entidad desconocida en Visitors: UnknownEntityException

Java SE 7 ha introducido otra pequeña mejora de esas que pasan desapercibidas en el diseño de las clases de la Language Model API: la creación de una nueva clase “excepción padre” que sirve para agrupar las diferentes excepciones arrojadas por los diferentes Visitors de la API cuando visitan una entidad desconocida: **UnknownEntityException**. El término “entidad” en este ámbito agruparía elementos, valores de anotaciones y tipos; es decir: todas las diferentes “entidades anotables/visitables” existentes en el lenguaje.

La introducción de la “excepción padre” **UnknownEntityException** es una mejora poco significativa a nivel práctico, pero importante a nivel de diseño, ya que proporciona un punto de anclaje para un concepto (el de visitar una “entidad desconocida”) que ya tenía 3 variantes distintas. Así pues, la relación entre la “excepción padre” y las hijas queda como sigue:



Error: MirroredTypeException al acceder vía reflexión a elementos Class []

La siguiente novedad introducida en Java SE se refiere a un bug corregido en la implementación de la API de reflexión. En la documentación oficial referida a las [mejoras en la implementación del lenguaje](#) se recoge la corrección de un bug importante relacionado con la implementación de los Mirrors, que ya había sido documentado por un usuario en un post de su blog dedicado a la [extracción de valores de tipo Class de los AnnotationMirrors](#), y que, justo antes de la conclusión, comentaba (traduciendo al castellano):

«Según los javadocs debería funcionar para **Class[]** arrojando una **MirroredTypesException** y proporcionando un método alternativo que retorna una lista de **TypeMirrors**. Sin embargo, no lo hace – simplemente arroja una **MirroredTypeException** para el primer elemento del array. Creo que cuando empieza a recorrer el array para poblarlo arroja la **MirroredTypeException** antes de tener la oportunidad de arrojar una **MirroredTypesException**».

Por tanto, habiendo sido ya corregido este bug en Java SE 7, cuando se accede mediante reflexión (usando **getAnnotation**) a un elemento de tipo **Class[]** ya sí se lleva a cabo por fin el comportamiento esperado: la implementación de **javac** arroja la **MirroredTypesException** con su lista correspondiente de **TypeMirror**.

17.2.3.- Novedades Java SE 7: Compiler Tree API.

La Compiler Tree API, por modelar el lenguaje Java desde el punto de vista sintáctico, también puede verse afectada por las novedades introducidas en el lenguaje. A continuación se listan las novedades introducidas en la Compiler Tree API en Java SE 7.

Compiler Tree API – Novedades Java SE 7	
Paquete com.sun.source.tree	
Visitor	Campo(s) / Método(s)
<code>Tree.Kind</code>	<code>ANNOTATION_TYPE</code> (para instancias de <code>ClassTree</code> que sean tipos anotación), <code>ENUM</code> (para instancias de <code>ClassTree</code> representando tipos enumerados), <code>INTERFACE</code> (para instancias de <code>ClassTree</code> representando tipos interfaces), <code>UNION_TYPE</code> (para instancias de <code>UnionTypeTree</code>).
<code>TreeVisitor</code>	<code>R visitUnionType(UnionTypeTree node, P p)</code>
<code>UnionTypeTree</code>	<code>List<? extends Tree> getTypeAlternatives()</code>
Paquete com.sun.source.util	
<code>Trees</code>	<code>public abstract String getDocComment(TreePath path)</code> <code>public abstract TypeMirror getLub(CatchTree tree)</code> <code>public abstract void printMessage(Diagnostic.Kind kind,</code> <code>CharSequence msg, Tree t, CompilationUnitTree root)</code>
<code>SimpleTreeVisitor</code>	<code>public R visitUnionType(UnionTypeTree node, P p)</code> (que por defecto llamará a <code>defaultAction</code>)
<code>TreeScanner</code>	<code>public R visitUnionType(UnionTypeTree node, P p)</code> (que por defecto escaneará todas las clases de la unión)
<code>TreePathScanner</code>	<code>public R visitUnionType(UnionTypeTree node, P p)</code> (heredado de <code>TreeScanner</code> , por def. escaneará todas las clases de la unión)

Como podemos ver, los cambios son los esperados al introducir el tipo unión de las sentencias multi-catch dentro del soporte de la Compiler Tree API. También se añaden algunos métodos interesantes a la clase `Trees` para recuperar los comentarios de documentación, así como para imprimir mensajes, que serán muy útiles para depuración.

Más sorprendente es, sin embargo, el hecho de que se incluyan las diferentes constantes de tipos de nodo especializados de `ClassTree` (`ANNOTATION_TYPE`, `ENUM`, e `INTERFACE` en `Tree.Kind`) para poder distinguir con más precisión el tipo de elemento específico que representa un nodo concreto, más que nada porque esto es algo que cualquiera esperaría que se hubiera incluido ya desde el principio.

Toda la información de referencia completa de la Compiler Tree API para Java SE 7 puede encontrarse en la URL de la documentación oficial de la API sita en la siguiente dirección:

<http://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/>

18.- NOVEDADES EN JAVA SE 8.

Las novedades en el lenguaje Java pueden afectar al procesamiento de anotaciones, ya que en él se manejan abstracciones de los elementos del lenguaje. Por ello, empezaremos con un apartado introductorio para explicar las novedades a nivel general introducidas en Java SE 8, seguido de un apartado con los cambios en las APIs relacionadas con el procesamiento, y finalizaremos explicando en detalle las novedades del procesamiento de anotaciones.

18.1.- Novedades generales en Java SE 8.

El 18 de Marzo de 2014, se liberaba **Java SE 8** (con el nombre clave o *codename* no oficial de “**Spider**”), desarrollada bajo la “especificación paraguas” [JSR 337](#), según el proceso establecido por el JCP en [OpenJDK](#). Hubo una novedad a nivel organizativo en el proceso de desarrollo de Java SE 8: los **JEP** ([JDK Enhancement Proposal](#), Propuesta de Mejora de la JDK), un modo más informal de agrupar propuestas antes de pasar a ser especificaciones JSR.

Las **mejoras más importantes introducidas en Java SE 8** son las siguientes:

- Lenguaje Java:
 - Código como datos: [Expr. lambda \(JSR 335, Project Lambda\)](#) y [Referencias a métodos](#).
 - Interfaces: [Métodos por defecto \(default](#) en métodos de interfaz).
 - Reflexión: Capacidad para [recuperar los nombres de los parámetros de un método](#).
 - Compilador: [Mejoras en la inferencia de tipos genéricos](#).
 - Anotaciones: [Anotaciones repetibles](#) y [Anotaciones sobre Tipos \(JSR 308\)](#).
- Colecciones: Nuevas [APIs de Streams](#) (“flujos”) de operaciones agregadas de estilo funcional.
- BD: [Mejoras JDBC](#) (JDBC 4.2, puente JDBC-ODBC eliminado). Incluida [Java DB 10](#) en la JDK.
- Utilidades: [Mejoras en internacionalización](#), [Date-Time API \(JSR 310\)](#), [Java Mission Control 5.3](#).
- Utilidades: Concurrencia: Muchísimas [novedades en la API de concurrencia](#).
- Utilidades: Ordenación de arrays en paralelo ([JEP 103](#)): [java.util.Arrays.parallelSort\(...\)](#).
- Utilidades: Manipulación en [Base64 \(JEP 135\)](#) y [aritmética de enteros sin signo](#).
- Utilidades: Librerías nativas [JNI \(Java Native Interface\)](#) estáticamente enlazadas ([JEP 178](#)).
- Modularización: [Perfiles compactos](#): subconjuntos de Java SE para dispositivos pequeños.
- Seguridad: [Mejoras de Seguridad](#): [TLS 1.1/1.2](#) activado por defecto y otras muchas.
- Herramientas: [Motor JavaScript Nashorn](#) para incluir código JS en aplicaciones Java.
- Herramientas: Comandos [jjs](#) (para invocar a Nashorn) y [jdeps](#) (análisis de dependencias).
- Herramientas: [java: Manual](#) reestructurado y con muchas nuevas opciones documentadas.
- Herramientas: [javac](#): Opciones [-parameters](#), [-h](#) y [comprobación de comentarios javadoc](#).
- Herramientas: [javadoc](#): [DocTree API \(JEP 105\)](#), [Javadoc Access API](#), [métodos por tipo y más](#).
- Herramientas: [apt](#): [Eliminada](#) la herramienta de procesamiento de anotaciones de J2SE 1.5.
- Rendimiento: [Strings](#): Constructor [String\(byte\[\], *\)](#) y método [String.getBytes\(\)](#).
- Rendimiento: Colecciones: Mejor rendimiento en [HashMap con múltiples colisiones de claves](#).
- Rendimiento: Eliminada [gen. permanente](#) en [HotSpot](#) para converger con [JRockit \(JEP 122\)](#).
- JavaFX: El comando [java](#) (y [javaw](#) en Windows) ahora también lanza aplicaciones JavaFX.
- JavaFX: [SwingNode](#), que permite [incluir componentes ligeros de Swing en JavaFX](#).
- JavaFX: Controles [DatePicker](#) y [TreeTableView](#), [tema Modena](#) y mejor renderización textual.
- JavaFX: Nuevas APIs de CSS ([javafx.css](#)) y JavaFX Printing API ([javafx.print](#)).
- JavaFX: Nuevas [capacidades 3D](#), [soporte mejorado para HTML5](#) y [otras muchas mejoras](#).

La lista completa de mejoras introducidas en Java SE 8 puede encontrarse en la página web oficial de Java SE 8 del portal OpenJDK [<http://openjdk.java.net/projects/jdk8/features>] y en OTN [<http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html>]

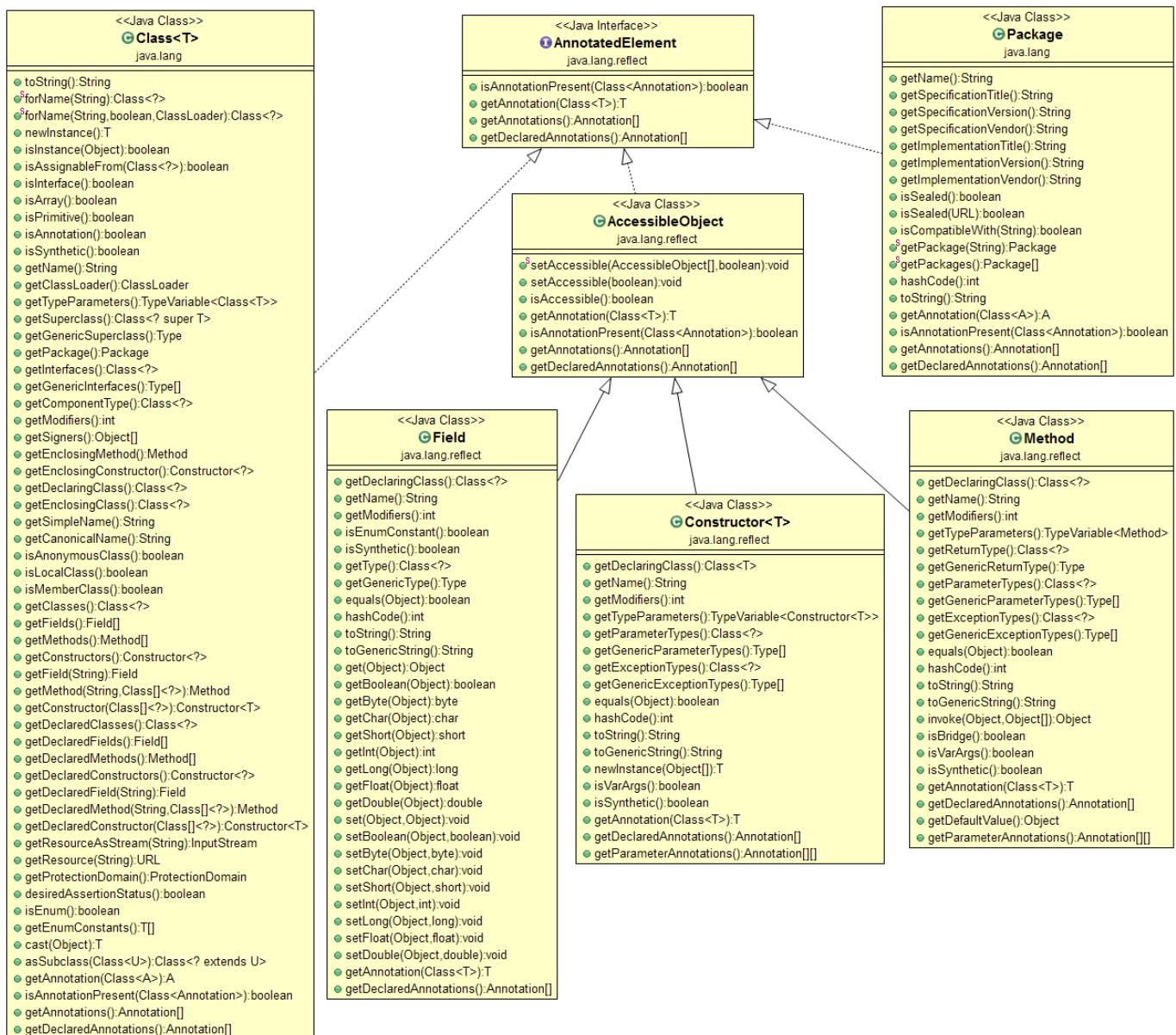
18.2.- Novedades Java SE 8 en APIs de procesamiento.

NOTA: A veces la novedad son los tipos (clases y/o interfaces) en sí mismos, otras veces lo es la inclusión de nuevos campos y/o métodos. Para diferenciarlo claramente, los elementos nuevos aparecerán en color **verde** en los siguientes apartados.

18.2.1.- Novedades Java SE 8: API de reflexión.

La API de reflexión (clases `java.lang.Class`, `java.lang.Package` y las clases del paquete `java.lang.reflect`) ha introducido muchos cambios en Java SE 8, aunque la [documentación oficial sobre sus mejoras](#) sólo refiere la recuperación de nombres de parámetros.

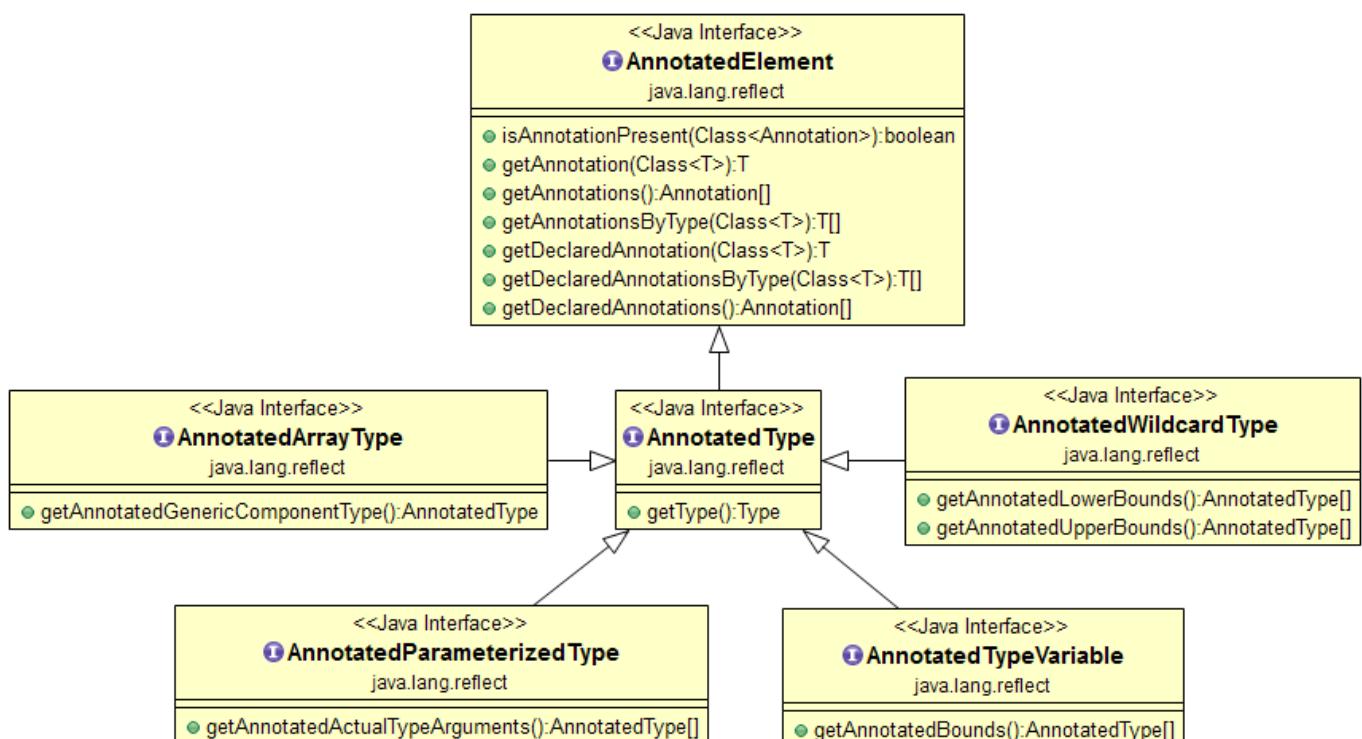
Los elementos del lenguaje Java sobre los que podía recuperarse información acerca de sus anotaciones vía API de reflexión son los que implementan la interfaz `AnnotatedElement`. Véase el [diagrama de clases implementadoras de AnnotatedElement en Java SE 7](#) (no Java SE 8):



Nuevas interfaces de anotaciones sobre usos de tipos

La mayoría de las novedades de la API de reflexión en Java SE 8 se deben principalmente a la especificación [JSR 308](#) (“Annotations on Java Types”), que introdujo en el lenguaje las anotaciones sobre tipos y las anotaciones repetibles. La interfaz `AnnotatedElement` fue actualizada para recuperar estos nuevos tipos de anotaciones con nuevos métodos e interfaces.

Las anotaciones sobre tipos se han añadido como interfaces que heredan de la nueva interfaz `AnnotatedType`, y que modelan los diferentes escenarios en que puede aparecer una anotación sobre el uso de un tipo: en un tipo array, un tipo parametrizado, una variable de tipo, o en un tipo comodín. `AnnotatedType` y sus nuevas subinterfaces a su vez heredan de `AnnotatedElement`, como muestra el siguiente **diagrama de clases que incluye las nuevas interfaces de anotaciones sobre tipos en Java SE 8:**



API de reflexión – Novedades Java SE 8

Paquete `java.lang.reflect` – `AnnotatedElement` y subinterfaces de anotaciones sobre tipos

Tipo	Campo(s) / Método(s)
<code>AnnotatedElement</code>	<code>default <T extends Annotation> T[] getAnnotationsByType(Class<T> annotationClass)</code> <code>default <T extends Annotation> T getDeclaredAnnotation(Class<T> annotationClass)</code> <code>default <T extends Annotation> T[] getDeclaredAnnotationsByType(Class<T> annotClass)</code>
<code>AnnotatedType</code>	<code>Type getType()</code>
<code>AnnotatedArrayType</code>	<code>AnnotatedType getAnnotatedGenericType()</code>
<code>AnnotatedParameterizedType</code>	<code>AnnotatedType[] getAnnotatedActualTypeArguments()</code>
<code>AnnotatedTypeVariable</code>	<code>AnnotatedType[] getAnnotatedBounds()</code>
<code>AnnotatedWildcardType</code>	<code>AnnotatedType[] getAnnotatedLowerBounds()</code> <code>AnnotatedType[] getAnnotatedUpperBounds()</code>

Uso de los métodos de la interfaz AnnotatedElement

En la documentación oficial de la interfaz [AnnotatedElement](#) se realizan una serie de definiciones para tener claro qué es lo que devuelve cada uno de sus métodos. Son las siguientes:

- **Directamente presente:** Si la anotación A está escrita en código sobre el elemento E: @A E
- **Indirectamente presente:** Si la anotación A es el tipo contenido de una anotación repetible R, y sobre el elemento E hay escrita una anotación R que contiene como valor una anotación A: @A E, que equivale a @R(@A) E, @R({@A}) E, y en general cualquier forma sintáctica que sirva para expresar un conjunto de anotaciones repetibles sobre E: @R(value={@A, @A, ..., @A}) E.
- **Presente:** A estará presente sobre E si está directamente presente o, si A es heredable y E es un elemento de tipo clase, si A está presente a través de la herencia de una superclase de E.
- **Asociada:** A estará asociada a E si está directa o indirectamente presente o, si A es heredable y E es elemento de tipo clase, si A está asociada a través de la herencia de una superclase de E.

En base a estas definiciones previas, la documentación oficial presenta una tabla con la clase de presencia que soportan cada uno de los métodos de la interfaz [AnnotatedElement](#):

API de reflexión – Novedades Java SE 8 – AnnotatedElement – Métodos y su Clase de presencia				
Método		Clase de presencia		
		Direct. Presente	Indirect. Presente	Presente
T	<code>getAnnotation(Class<T>)</code>			x
Annotation[]	<code>getAnnotations()</code>			x
T[]	<code>getAnnotationsByType(Class<T>)</code>			x
T	<code>getDeclaredAnnotation(Class<T>)</code>	x		
Annotation[]	<code>getDeclaredAnnotations()</code>	x		
T[]	<code>getDeclaredAnnotationsByType(Class<T>)</code>	x	x	

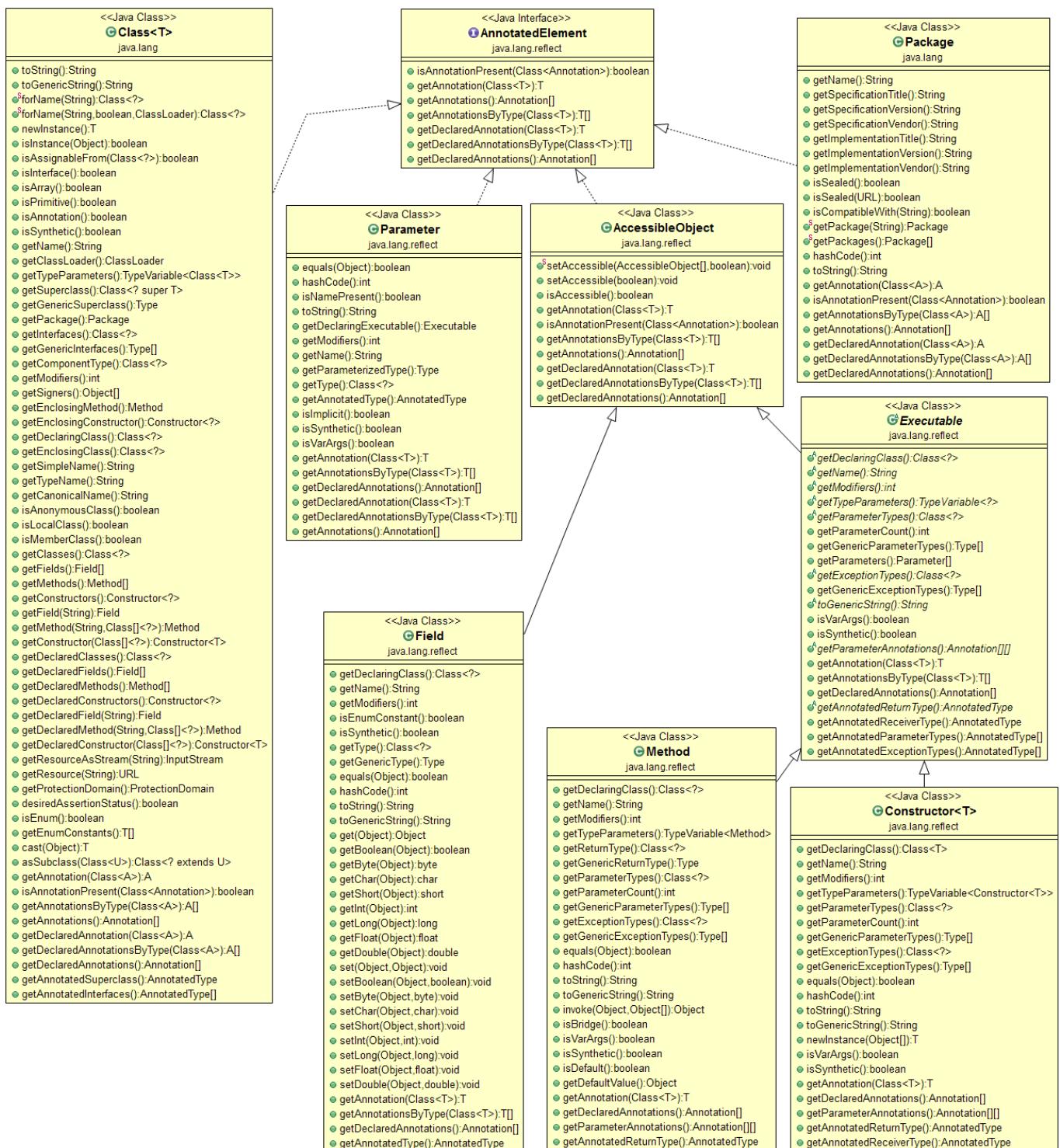
Las definiciones de las clases de presencia se podrían reformular en base únicamente a tres capacidades: Presencia directa, Soporte de herencia, y Soporte de anotaciones repetibles:

- **Directamente presente:** Presencia directa.
- **Indirectamente presente:** Soporte de anotaciones repetibles.
- **Presente:** Presencia directa + Soporte de herencia.
- **Asociada:** Presencia directa + Soporte de anotaciones repetibles + Soporte de herencia.

API de reflexión – Novedades Java SE 8 – AnnotatedElement – Métodos y sus Capacidades				
Método		Capacidades		
		Presencia Directa	Soporte de Herencia	Anotaciones Repetibles
T	<code>getAnnotation(Class<T>)</code>	✓	✓	
Annotation[]	<code>getAnnotations()</code>	✓	✓	
T[]	<code>getAnnotationsByType(Class<T>)</code>	✓	✓	✓
T	<code>getDeclaredAnnotation(Class<T>)</code>	✓		
Annotation[]	<code>getDeclaredAnnotations()</code>	✓		
T[]	<code>getDeclaredAnnotationsByType(Class<T>)</code>	✓		✓

Nuevas clases implementadoras de `AnnotatedElement`

Hay también alguna novedad entre los elementos del lenguaje Java sobre los que puede recuperarse información acerca de sus anotaciones vía API de reflexión (es decir: los elementos cuyas clases reflexivas implementan la interfaz `AnnotatedElement`). Y es que se añaden dos nuevas clases: `Parameter` para la nueva reflexión sobre los parámetros de los métodos, y `Executable`, que agrupa constructores y métodos (mimetizando el `ExecutableElement` de la Language Model API). El siguiente es el **diagrama de clases implementadoras de `AnnotatedElement` en Java SE 8 (hay que prestar especial atención a todos los métodos que devuelven `AnnotatedType`, que suelen empezar por `getAnnotated`)**:



Además de los nuevos métodos comunes de la interfaz **AnnotatedElement**, las clases que la implementan (es decir, las clases que representan elementos del lenguaje que pueden ser anotados), han introducido también nuevos métodos propios fundamentales, principalmente para poder recuperar instancias de las nuevas anotaciones sobre usos de tipos cuando estemos descubriendo las anotaciones sobre tipos cuando realicemos procesamiento de anotaciones a través de la API de reflexión:

API de reflexión – Novedades Java SE 8 – Clases implementadoras de AnnotatedElement	
Paquete <code>java.lang.reflect</code> (excepto <code>Class</code> y <code>Package</code> , que son de <code>java.lang</code>)	
Clases	Método(s) (no se incluyen los nuevos métodos de la interfaz AnnotatedElement)
<code>Class</code>	<pre>public String toGenericString() public String get TypeName() public AnnotatedType getAnnotatedSuperclass() public AnnotatedType[] getAnnotatedInterfaces()</pre>
<code>Package</code>	No define ningún método nuevo (a parte de los de AnnotatedElement).
<code>Parameter</code>	Todos los métodos, ya que es una clase completamente nueva.
<code>Field</code>	<pre>public AnnotatedType getAnnotatedType()</pre>
<code>Executable</code>	<p>NOTA: A diferencia de <code>Parameter</code>, no todos los métodos de <code>Executable</code> son nuevos. Muchos pertenecían a <code>Method</code> y <code>Constructor</code>, de las que ahora es superclase.</p> <pre>public AnnotatedType[] getAnnotatedExceptionTypes() public AnnotatedType[] getAnnotatedParameterTypes() public AnnotatedType getAnnotatedReceiverType() public abstract AnnotatedType getAnnotatedReturnType() public int getParameterCount() public Parameter[] getParameters()</pre>
<code>Method</code>	<pre>public AnnotatedType getAnnotatedReturnType() (en Executable es abstract) public boolean isDefault()</pre>
<code>Constructor</code>	<pre>public AnnotatedType getAnnotatedReturnType() (en Executable es abstract)</pre>

IMPORTANTE: En cuanto al procesamiento de anotaciones sobre usos de tipos en Java SE 8, como ya hemos comentado, son de especial interés todos los métodos nuevos que devuelven instancias de la interfaz **AnnotatedType** (que es la que modela los usos de tipos anotados), ya sean instancias únicas o como arrays **AnnotatedType**[], puesto que será principalmente con estos métodos con los cuales habremos de recuperar las instancias que nos permitirán procesar las nuevas anotaciones sobre usos de tipos.

Asimismo, se aprecian los métodos nuevos de **Executable** que modelan la reflexión sobre los parámetros de los elementos de código ejecutables (esto es: métodos y constructores): `getParameterCount()` y `getParameters()`. Este último devuelve un array de la nueva clase **Parameter**, que contiene toda la información reflexiva referente a los parámetros.

Finalmente, la clase **Method** introduce con su nuevo método `isDefault()` un método que nos permite saber si el método que estamos procesando se trata de uno de los nuevos métodos por defecto introducidos en Java SE 8 para las interfaces.

18.2.2.- Novedades Java SE 8: Language Model API.

Java SE 8 introdujo muchas novedades en la Language Model API debido principalmente a la introducción de las nuevas “anotaciones sobre tipos” dadas según la especificación [JSR 308](#) (“Annotations on Java Types”). A continuación enumeramos las novedades más importantes.

Nuevo concepto de “construcción anotada”

Java SE 8 introduce el concepto de “construcción anotada” (“annotated construct”). El término “construcción” puede ser tanto un elemento como un tipo. Las anotaciones sobre elementos anotan una declaración, mientras que las anotaciones sobre tipos anotan un uso específico del nombre del tipo. Las construcciones anotadas se modelan con la superinterfaz `AnnotatedConstruct`.

`AnnotatedConstruct` es superinterfaz de `Element` (raíz de la jerarquía de elementos) y de `TypeMirror` (raíz de la jerarquía de tipos). Esto quiere decir que toda entidad anotable será instancia de `AnnotatedConstruct` y, así pues, también dispondrá de los métodos definidos en dicha superinterfaz.

Finalmente, en Java SE 8 se ha añadido una nueva interfaz para los “[tipos intersección](#)”: `IntersectionType`. El modelado de estos tipos estaba en Java SE 6, sólo que se hacía usando `DeclaredType` donde la única clase del tipo intersección se modelaba como superclase y todos los tipos interfaz como superinterfaces. Ahora se puede utilizar este nuevo tipo especializado.

Language Model API – Novedades Java SE 8	
Paquete <code>javax.lang.model</code>	
Tipo	Campo(s) / Método(s)
<code>SourceVersion</code>	<code>RELEASE_8</code> [campo que modela la versión de Java SE 8]
<code>AnnotatedConstruct</code>	<code><A extends Annotation> A getAnnotation(Class<A> annotationType)</code> <code><A extends Annotation> A[] getAnnotationsByType(Class<A> annotType)</code> <code>List<? extends AnnotationMirror> getAnnotationMirrors()</code>
Paquete <code>javax.lang.model.element</code>	
Tipo	Campo(s) / Método(s)
<code>Modifier</code>	<code>DEFAULT</code> [campo que modela el modificador de los “ métodos por defecto ”]
<code>ExecutableElement</code>	<code>TypeMirror getReceiverType()</code> <code>boolean isDefault()</code>
Paquete <code>javax.lang.model.type</code>	
Tipo	Campo(s) / Método(s)
<code>ExecutableType</code>	<code>TypeMirror getReceiverType()</code>
<code>TypeKind</code>	<code>INTERSECTION</code> [campo que modela los “ tipos intersección ”: T1 & ... & Tn]
<code>IntersectionType</code>	<code>List<? extends TypeMirror> getBounds()</code> [para el ejemplo de arriba devolvería la lista T1, ..., Tn]
<code>TypeVisitor</code>	<code>R visitIntersection(IntersectionType t, P p)</code>
Paquete <code>javax.lang.model.util</code>	
Tipo	Campo(s) / Método(s)
<code>Elements</code>	<code>boolean isFunctionalInterface(TypeElement type)</code>

Novedades relacionadas con los Visitor de Elementos en Java SE 8

Los Visitor de Elementos en Java SE 8 no introducen ningún cambio, ya que no se han introducido nuevos tipos de elementos. Todas las clases Visitor8 de elementos se limitan simplemente a heredar de las respectivas clases Visitor7 y a definir sus constructores, pero ni añaden ni redefinen ni un sólo método. En este caso, quizás hubiera sido mejor no definir nuevas clases para los Visitor de elementos de Java SE 8 y confundir más a los desarrolladores, pero sabemos que definir nuevas clases Visitor para cada nueva versión del lenguaje Java es así por norma de los diseñadores de la Language Model API.

Language Model API – Novedades Java SE 8 – Visitors de Elementos	
Paquete javax.lang.model.util	
Visitor	Método(s)
<code>AbstractElementVisitor8</code>	No define ningún método nuevo.
<code>SimpleElementVisitor8</code>	No define ningún método nuevo.
<code>ElementKindVisitor8</code>	No define ningún método nuevo.
<code>ElementScanner8</code>	No define ningún método nuevo.

Novedades de los Visitor de Valores de Anotaciones en Java SE 8

El caso de los Visitor de Valores de anotaciones en Java SE 8 es idéntico al de los Visitor de elementos: no hay novedad alguna. No obstante, también se han definido las clases rutinarias para cada uno de los tipos de Visitor de Valores de anotaciones. Estas clases, al igual que las implementaciones de Visitor de elementos, se limitan a simplemente a heredar de sus versiones de Java SE 7 y definir los constructores necesarios. No siendo necesarias en absoluto, podían habérselas ahorrado, pero, como ya hemos dicho, esto es norma de los diseñadores de la API.

Language Model API – Novedades Java SE 8 – Visitors de Valores de Anotaciones	
Paquete javax.lang.model.util	
Visitor	Método(s)
<code>AbstractAnnotationValueVisitor8</code>	No define ningún método nuevo.
<code>SimpleAnnotationValueVisitor8</code>	No define ningún método nuevo.

Novedades relacionadas con los Visitor de Tipos en Java SE 8

Aunque ya era modelado desde Java SE 6 a través de la interfaz `DeclaredType`, los Visitor de Tipos añaden una nueva interfaz especializada para modelar el denominado “[tipo intersección](#)”: `IntersectionType`, así como la correspondiente constante `TypeKind.INTERSECTION`, por lo que la interfaz `TypeVisitor` también añade un método `visitIntersection`, lo que hace que todos los Visitor de tipos deban ahora implementar dicho método.

IMPORTANTE: Al incluir el nuevo `visitIntersection` en `TypeVisitor`, los Visitor de tipos de Java SE 6/7 también implementarán dicho método, pero, como es lógico, llamarán a `visitUnknown`. Aquí es donde se hace imprescindible que, si se implementó un `TypeVisitor` para Java SE 6/7, haber redefinido `visitUnknown` para no ejecutar su implementación por defecto, que arroja `UnknownTypeException`. Si no se redefinió `visitUnknown`, nuestro `TypeVisitor` funcionaría correctamente en Java SE 6/7, pero NO en Java SE 8, fallando al visitar un tipo intersección, ya que se llamaría a `visitUnknown` y este arrojaría la excepción, por lo que itendríamos un procesador válido sólo para Java SE 6/7!, algo inaceptable.

Language Model API – Novedades Java SE 8 – Visitors de Tipos	
Paquete javax.lang.model.util	
Visitor	Método(s)
<code>AbstractTypeVisitor6</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a visitUnknown)</code>
<code>AbstractTypeVisitor7</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a visitUnknown)</code>
<code>AbstractTypeVisitor8</code>	<code>public abstract R visitIntersection(IntersectionType t, P p)</code>
<code>SimpleTypeVisitor6</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a visitUnknown)</code>
<code>SimpleTypeVisitor7</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a visitUnknown)</code>
<code>SimpleTypeVisitor8</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a defaultAction)</code>
<code>ElementKindVisitor6</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a visitUnknown)</code>
<code>ElementKindVisitor7</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a visitUnknown)</code>
<code>ElementKindVisitor8</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a defaultAction)</code>
<code>TypeKindVisitor6</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a visitUnknown)</code>
<code>TypeKindVisitor7</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a visitUnknown)</code>
<code>TypeKindVisitor8</code>	<code>public R visitIntersection(IntersectionType t, P p)</code> <code>(que por defecto llamará a defaultAction)</code>

NOTA: `visitIntersection` es `abstract` sólo para `AbstractTypeVisitor8` y, sin embargo, no lo es para los anteriores. Esto es así porque las clases abstractas de las versiones anteriores de la plataforma Java pueden dar la implementación por defecto típica para `visitIntersection`, que será la de llamar a `visitUnknown`, mientras que para el Visitor de Java SE 8 el tipo intersección no es un tipo nuevo ni distinto de cualquiera de los demás, por lo que no se puede fijar un comportamiento por defecto, si no que se deja la definición de dicha lógica al implementador dejándolo como `abstract`. Los Visitor concretos de tipos de Java SE 8 que heredan de `AbstractTypeVisitor8` ya sí que implementan el método siguiendo cada uno de ellos el contrato común de las demás implementaciones sencillas, que es llamar al método `defaultAction`.

18.2.3.- Novedades Java SE 8: Compiler Tree API.

La Compiler Tree API introdujo una buena cantidad de novedades en Java SE 8 debido a las muchas novedades incluidas en el lenguaje Java para dicha versión: soporte de árboles sintácticos de los comentarios de documentación para javadoc, así como actualizaciones del soporte de árboles sintácticos para los nuevos elementos del lenguaje, como las anotaciones sobre tipos, las expresiones lambda y las referencias a métodos.

Para no mezclar ambas funcionalidades de la Compiler Tree API, que conceptualmente irían más en paralelo que en conjunto, trataremos dichas novedades en apartados independientes.

ÁRBOLES SINTÁCTICOS DE COMENTARIOS DE DOCUMENTACIÓN

Una gran novedad de la Compiler Tree API en Java SE 8 es la introducción de un nuevo paquete `com.sun.source.doctree`, así como de nuevas clases en `com.sun.source.util` para poder manejar como árboles sintácticos abstractos los elementos de los comentarios de documentación de `javadoc`.

Los árboles sintácticos de comentarios de documentación permiten el procesamiento fácil y detallado de dichos comentarios, tanto dentro de un procesador de anotaciones como en cualquier otro tipo de aplicación Java.

Antes de tener estas nuevas facilidades, era posible acceder únicamente al texto de un comentario de documentación a través del método `Elements.getDocComment(Element e)` de JSR 269, y de `Trees.getDocComment(TreePath path)` proporcionado por la propia Compiler Tree API (a partir de Java SE 7), pero ahora incluso se puede procesar dicha información de forma altamente estructurada en árboles sintácticos, por lo que, desde Java SE 8, sería sencillo, por ejemplo, que desarrolladores independientes pudieran implementar su propia versión alternativa de `javadoc`.

Compiler Tree API – Novedades Java SE 8 – Árboles sintácticos de comentarios de documentación	
Paquete com.sun.source.doctree	
Tipos	Campo(s) / Método(s)
<code>Todos los tipos del paquete</code>	<code>El nuevo paquete doctree permite modelar como árboles sintácticos los elementos de los comentarios de documentación de javadoc.</code>
Paquete com.sun.source.util	
Tipos	Campo(s) / Método(s)
<code>DocSourcePositions</code>	<code>Todos los métodos. Hereda de SourcePositions.</code>
<code>DocTreePath</code>	<code>Todos los métodos.</code>
<code>SimpleDocTreeVisitor</code>	<code>Todos los métodos. Implementa DocTreeVisitor.</code>
<code>DocTreeScanner</code>	<code>Todos los métodos. Implementa DocTreeVisitor.</code>
<code>DocTreePathScanner</code>	<code>Todos los métodos. Hereda de DocTreeScanner.</code>
<code>DocTrees</code>	<code>Todos los métodos. Hereda de Trees.</code>

ANOTACIONES SOBRE TIPOS, LAMDBAS Y REFERENCIAS A MÉTODOS

En la parte clásica de árboles sintácticos sobre elementos de código se añaden clases para modelar todas las nuevas capacidades introducidas en el lenguaje Java en Java SE 8: las nuevas anotaciones sobre usos de tipos y tipos parámetro, las expresiones lambda y las referencias a métodos. También se introduce soporte explícito para el tipo intersección. Se añaden, por tanto, todos los métodos correspondientes a las nuevas interfaces a las interfaces Visitor. Finalmente, se introduce la nueva clase **Plugin**, que modela plugins (no procesadores de anotaciones) para registrar un manejador de eventos de compilación que permite realizar tareas vinculadas con los eventos de inicio y fin de la compilación de cada unidad de compilación.

Compiler Tree API – Novedades Java SE 8 – Árboles sintácticos de elementos de código	
Paquete com.sun.source.tree	
Tipos	Campo(s) / Método(s)
AnnotatedTypeTree	<code>List<? extends AnnotationTree> getAnnotations()</code> [este es el método que recupera las nuevas anotaciones sobre usos de tipos] <code>ExpressionTree getUnderlyingType()</code>
IntersectionTypeTree	<code>List<? extends Tree> getBounds()</code>
LambdaExpressionTree	<code>LambdaExpressionTree.BodyKind getBodyKind()</code> <code>List<? extends VariableTree> getParameters()</code> <code>Tree getBody()</code>
LambdaExpressionTree.BodyKind	<code>EXPRESSION</code> [expresiones lambda, cuyo cuerpo es una expresión] <code>STATEMENT</code> [sentencias lambda, cuyo cuerpo es un bloque de código]
MemberReferenceTree	<code>MemberReferenceTree.ReferenceMode getMode()</code> <code>ExpressionTree getQualifierExpression()</code> <code>Name getName()</code> <code>List<? extends ExpressionTree> getTypeArguments()</code>
MemberReferenceTree.ReferenceMode	<code>INVOKE</code> [referencias a métodos] <code>NEW</code> [referencias a constructores]
MethodTree	<code>VariableTree getReceiverParameter()</code> [devuelve una referencia a la instancia de la variable sobre la que se ha invocado el método]
TreeVisitor	<code>R visitAnnotatedType(AnnotatedTypeTree node, P p)</code> <code>R visitIntersectionType(IntersectionTypeTree node, P p)</code> <code>R visitLambdaExpression(LambdaExpressionTree node, P p)</code> <code>R visitMemberReference(MemberReferenceTree node, P p)</code>
TypeParameterTree	<code>List<? extends AnnotationTree> getAnnotations()</code> [este es el método que recupera las nuevas anotaciones sobre parámetros de tipo]
Paquete com.sun.source.util	
Tipos	Campo(s) / Método(s)
Plugin	<code>String getName()</code> <code>void init(JavacTask task, String... args)</code>
SimpleTreeVisitor, TreeScanner y TreePathScanner (todos los Visitor)	<code>public R visitAnnotatedType(AnnotatedTypeTree node, P p)</code> <code>public R visitIntersectionType(IntersectionTypeTree node, P p)</code> <code>public R visitLambdaExpression(LambdaExpressionTree node, P p)</code> <code>public R visitMemberReference(MemberReferenceTree node, P p)</code>

Toda la información de referencia completa de la Compiler Tree API para Java SE 8 puede encontrarse en la URL de la documentación oficial de la API sita en la siguiente dirección:

<http://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/>

18.3.- Novedades del procesamiento en Java SE 8.

18.3.1.- Eliminación definitiva de apt (JSR 269 / JEP 117).

Aunque la herramienta **apt** quedó anticuada incluso ya desde el mismo momento de la introducción de la especificación JSR 269, fue [deprecada formalmente en Java SE 7](#). Y fue eliminada de forma definitiva de la distribución oficial de la plataforma Java con la llegada de Java SE 8 a través de la aprobación de una propuesta de mejora de la JDK: la [JEP 117](#) “Remove the Annotation-Processing Tool (**apt**)”, completada en 2012 con la eliminación definitiva de todo rastro de la presencia de **apt** en la distribución oficial de la JDK de Java SE 8.

Antes de su deprecación y eliminación definitiva, en 2006, la Mirror API fue [liberada como open source](#) bajo licencia BSD. En 2010, su dominio fue desmantelado y sólo quedan los [ficheros de código fuente](#) de la API publicados por Joseph Darcy, su ingeniero jefe, en su blog.

El [blog de Joseph Darcy](#), a veces referido también por su diminutivo como Joe Darcy, se constituye como el punto de referencia informativa más importante acerca de procesamiento de anotaciones en Java, por el evidente motivo de ser él el principal responsable del diseño de toda la tecnología de procesamiento de anotaciones. A través de varios de sus posts puede rastrearse el recorrido histórico de **apt** y la Mirror API. Se citan a continuación como referencia estos:

- 2006/01/10 [apt mirror API Open Sourced](#) (“La Mirror API de apt se hace open source”)
- 2009/07/27 [An apt replacement](#) (incluye guía de correspondencias Mirror API ↔ Language Model API)
- 2010/11/15 [Original apt API files](#) (“Ficheros originales de la API de **apt**”)
- 2011/12/02 [An apt ending draws nigh](#) (“El final de **apt** pinta cercano”)
- 2012/02/10 [The passing of apt](#) (“La muerte de **apt**”)

Es importante tener en cuenta que, al ser **apt** y la Mirror API tecnologías Java basadas en versiones anteriores de la plataforma Java, aunque hayan sido deprecadas e incluso suprimidas de la distribución general de Java SE 8, es posible aún trabajar con ellas hasta Java SE 7. Se pueden usar los ficheros de código fuente de los ficheros apt-mirror-api.tar o apt-mirror-api.zip para generar una librería **apt-mirror-api.jar**, o puede descargarse de internet.

Teniendo la librería **apt-mirror-api.jar**, ya no será necesario para poder compilar el uso de la librería estándar **tools.jar** de la JDK (normalmente ubicada en el subdirectorio \lib\tools.jar de la instalación de la JDK). Es en tools.jar donde se incluyen todas las clases relacionadas con las herramientas de compilación y, por tanto, **es precisamente en la librería tools.jar donde ya no están las clases de la Mirror API, desde Java SE 8**.

Así pues, usando el JDK de Java SE 8, todos los **import** de clases de la Mirror API, ya no serán reconocidos por defecto en la plataforma Java, como le ocurrió, por ejemplo, a [este desarrollador](#). Eso se podría solucionar incluyendo la librería **apt-mirror-api.jar** en nuestro CLASSPATH, con lo que **podríamos seguir compilando y escribiendo procesadores de anotación J2SE 1.5 incluso en código Java SE 8**. No obstante, dicho código no podría sin embargo ser procesado por **apt**, ya que, aunque el código Java antiguo sí podría ser compilable por una versión más moderna del compilador, la implementación de **apt** más reciente, que es la incluida con Java SE 7, sólo es capaz de procesar código fuente compilado como mucho hasta con el compilador de la propia Java SE 7, como, por otra parte, no podía ser de otra manera.

Por tanto, el uso de **apt** tiene realmente los días contados, y es que, como decimos, con la última implementación de **apt**, la incluida con Java SE 7, no será posible procesar clases con un código fuente con un nivel >7 (versión 51 del formato de fichero de clases). En caso de intentar algo semejante, **apt** arrojará un error del tipo **UnsupportedClassVersionError**. El siguiente es el mensaje de error generado por **apt** 1.7.0 al tratar de procesar las anotaciones de una clase escrita con un nivel de código 8 (versión 52 del formato de fichero de clases). Nótese además como en la cabecera de la herramienta se incluye un aviso recomendando la migración a la API estándar de procesamiento de anotaciones de la especificación JSR 269.

```
apt 1.7.0_05

warning: The apt tool and its associated API are planned to be removed in the next major JDK
release. These features have been superseded by javac and the standardized annotation processing
API, javax.annotation.processing and javax.lang.model. Users are recommended to migrate to the
annotation processing features of javac; see the javac man page for more information.
Problem encountered during annotation processing; see stacktrace below for more information.
java.lang.UnsupportedClassVersionError: anotaciones/ejemplos/procesamiento/java5
/holaMundo/AnotacionHolaMundoProcessorFactory : Unsupported major.minor version 52.0
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:791)
    at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:449)
    at java.net.URLClassLoader.access$100(URLClassLoader.java:71)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:423)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:356)
    at com.sun.tools.apt.comp.Apt.main(Apt.java:306)
    at com.sun.tools.apt.main.AptJavaCompiler.compile(AptJavaCompiler.java:270)
    at com.sun.tools.apt.main.Main.compile(Main.java:1127)
    at com.sun.tools.apt.main.Main.compile(Main.java:989)
    at com.sun.tools.apt.Main.processing(Main.java:113)
    at com.sun.tools.apt.Main.process(Main.java:61)
    at com.sun.tools.apt.Main.main(Main.java:52)
warning: .\bin\anotaciones\ejemplos\procesamiento\java5\holaMundo\AnotacionHolaMundo.class: major
version 52 is newer than 51, the highest major version supported by this compiler.
It is recommended that the compiler be upgraded.
```

Evidentemente, se recomienda, por activa y por pasiva, una inmediata migración de todos los procesadores de anotaciones J2SE 1.5 a código equivalente JSR 269. Joseph Darcy, en la entrada de su blog titulada “[An apt replacement](#)”, hacía la siguiente recomendación:

« Como ingeniero jefe de ambas, **apt** y JSR 269, la API de JSR 269 así como su experiencia de uso deberían ser uniformemente mejores que con **apt**; la nueva API es más fácil de usar, más flexible, y debería ejecutarse más rápido también. Recomiendo incondicionalmente transicionar a la API de JSR 269 y **javac** (o su [Compiler API](#)) para todas sus necesidades de procesamiento de anotaciones. »

18.3.2.- Anotaciones repetibles (JSR 269 MR 2 / JEP 120).

Una novedad de Java SE 8 incluida como parte de la [JSR 269 MR 2](#) (“Maintenance Review 2”, es decir: “Revisión de Mantenimiento 2”) y vehiculada a través de la [JEP 120](#), que quizás pueda parecer de menor calado pero que añadirá mucha comodidad al uso de las anotaciones por parte de los usuarios, es la introducción de [anotaciones repetibles](#).

Las anotaciones repetibles permiten a los usuarios de las mismas anotar un elemento de código con el mismo tipo anotación un número indeterminado de veces. Por ejemplo, imaginemos que queremos controlar los roles de usuario que pueden ejecutar un determinado método, gracias a las anotaciones repetibles podemos escribir lo siguiente:

```
@RolUsuario("dependiente")
@RolUsuario("administrativo")
@RolUsuario("director")
public void metodoSeguro() { ... }
```

Para más detalles, puede consultarse el apartado “Anotaciones repetibles (Java SE 8+)”, donde se explica cómo diseñar tipos anotación repetibles, e incluso cómo convertir en repetibles tipos anotación que originalmente no lo eran.

18.3.3.- Anotaciones sobre tipos y proc. variables locales (JSR 308 / JEP 104).

Java SE 8 incluye la especificación [JSR 308](#) (“Annotations on Java Types”), vehiculada a través del [JEP 104](#), para soportar anotaciones sobre tipos Java.

Soportar anotaciones sobre elementos de tipos Java significa que se pueden escribir anotaciones allí donde la sintaxis del lenguaje permita escribir un tipo.

Además de lo anterior, la JSR 308 viene a resolver finalmente algunos problemas pendientes del soporte de anotaciones que se habían venido arrastrando hasta ahora.

Uno de los cambios más importantes que trae consigo JSR 308 es la actualización del formato de los ficheros compilados de clase con la información necesaria para el procesamiento de anotaciones sobre variables locales.

Dada la importancia de la especificación JSR 308, dedicaremos el siguiente apartado del presente manual a describir sus cambios en mayor profundidad.

Finalmente, los autores de la especificación JSR 308, en base a sus nuevas capacidades, han creado el [Checker Framework](#). Este framework no está incluido en el canon oficial de Java por el JCP, pero pone las bases para un uso muy interesante del procesamiento de anotaciones: la creación de “sistemas de comprobación de tipos extendida” para la comprobación avanzada de errores o warnings de compilación que el compilador Java estándar no puede reconocer. Esta nueva funcionalidad puede cobrar una enorme relevancia en un futuro próximo, ya que puede llegar a reducir a la mínima expresión los errores en tiempo de ejecución, ya que una gran cantidad de ellos podrían ser inferidos y corregidos en tiempo de compilación sin necesidad de ejecutar código. Para más información, véase el apartado monográfico del presente manual dedicado al Checker Framework.

19.- ANOTACIONES SOBRE TIPOS (JSR 308).

En este apartado profundizaremos en la especificación [JSR 308](#) (“Annotations on Java Types”, es decir: “**Anotaciones sobre tipos Java**”), que ya hemos mencionado múltiples veces a lo largo del presente manual por su gran trascendencia en el ámbito del procesamiento de anotaciones, debido a la gran cantidad de novedades que incluyó en todos los aspectos del procesamiento de anotaciones.

La **principal novedad de JSR 308** es la introducción de las **anotaciones sobre los usos de los tipos Java**, lo cual otorga una enorme potencia expresiva a las anotaciones, ya que, por regla general, ahora se pueden escribir anotaciones en todas las ubicaciones del código donde se pueden escribir tipos. Esto permite introducir un nuevo “sistema de tipos mejorado” que podrá utilizarse para dar mayores capacidades de análisis y control de errores al compilador Java, como veremos a continuación.

Otra novedad importante es que JSR 308 define una ampliación de la información almacenada en los ficheros de clase para permitir solventar las carencias del procesamiento de anotaciones en versiones anteriores de la plataforma Java. En base a esa nueva información añadida, JSR 308 permite introducir en Java SE 8 la capacidad de realizar **procesamiento de anotaciones sobre variables locales y sobre parámetros de tipo**.

19.1.- Un sistema de tipos mejorado.

El sistema de tipos que utiliza el compilador Java es similar al de los lenguajes que se han creado en las últimas décadas. Veamos por ejemplo la siguiente sentencia:

```
int i = "hello"; // error de tipo
```

Efectivamente, el sistema de tipos de Java permite detectar errores sencillos como el anterior, en el que intentamos asignar una cadena de caracteres a una variable de tipo entero. No obstante, la experiencia práctica de los últimos años ha demostrado que esta comprobación de tipos básica no es suficiente para eliminar la mayoría de los errores de ejecución más usuales.

Así pues, aunque el sistema de tipos de Java previene algunos de los errores más triviales y comunes, no previene los suficientes. Por ejemplo, imaginemos un error más complejo de detectar, que ni siquiera sea sintáctico, si no semántico. Es decir: escribir algo sintácticamente correcto, pero que sea claramente erróneo, como añadir un valor a una lista inmutable:

```
// añadimos un valor a una lista inmutable -> se arroja una UnsupportedOperationException
Collections.emptyList().add("un valor cualquiera");
```

Este error es evidente y el compilador debería ser capaz de detectarlo, pero el sistema de tipos clásico de Java no ofrece la suficiente información como para que el compilador pueda llevar a cabo dicha detección.

Supongamos otro ejemplo más complicado de ver incluso en el código fuente, pero igual o más pernicioso, puesto que no arroja ninguna excepción y es, por tanto, más difícil de detectar:

```
HashMap hashMap = new HashMap();
Date fecha = new Date();
hashMap.put(fecha, "Fecha actual");
fecha.setYear(2000);
hashMap.put(fecha, "Fecha del año 2000");
// aquí el mapa hash ha sido corrompido, al cambiar,
// debido a los efectos laterales, el objeto de fecha
// asociado la primera clave, que debería ser una fecha actual
```

Si dispusiéramos de una anotación `@Inmutable` que pudiéramos aplicar sobre el tipo de la variable `fecha`, esta ya no sería de tipo `Date` a secas, si no `@Inmutable Date`. Si el compilador fuera capaz de comprobar ese nuevo “sistema de tipos mejorado” mediante algún añadido (como, por ejemplo, un procesador de anotaciones JSR 269), ¡¡ipodríamos detectar el error de la corrupción del mapa hash y otros muchos más en tiempo de compilación!!!

```
HashMap hashMap = new HashMap();
@Inmutable Date fecha = new Date();
hashMap.put(fecha, "Fecha actual");
fecha.setYear(2000); // !!!ERROR DE COMPILACIÓN!!! :-
hashMap.put(fecha, "Fecha del año 2000");
```

Esto es lo que hace posible la especificación JSR 308. Así pues, las anotaciones sobre tipos adquieren ahora nuevas e importantes características:

- 1) Especifican información declarativa sobre los elementos del código fuente.
- 2) La información especificada por las anotaciones puede ser analizada de manera automática.

Por supuesto, JSR 308 sólo define la sintaxis de las anotaciones sobre tipos, así que la semántica concreta de las anotaciones sobre tipos dependerá de que se acoplen los respectivos procesadores de anotaciones al compilador Java, ya que, si no, las anotaciones por sí mismas, como ya sabemos, no tienen en absoluto efecto alguno sobre el código.

Gracias a las nuevas funcionalidades introducidas por la especificación JSR 308, y si creamos las anotaciones sobre tipos adecuadas (con sus correspondientes procesadores de anotaciones) y anotamos correctamente nuestro código, podríamos llegar a poder incluso garantizar que nuestro software no tiene ningún defecto ya desde el momento de compilarlo, sin tener siquiera que ejecutarlo o depurarlo. Esto puede parecer difícil de lograr pero, con un uso adecuado de las funcionalidades de comprobación del nuevo sistema de tipos mejorado, es perfectamente posible, al menos si no para todos los tipos de errores posibles, sí para un buen número de ellos. De esta forma, gracias a estos nuevos tipos de “controles preventivos”, los desarrolladores podrían incrementar la productividad y la calidad de sus aplicaciones.

El uso más habitual de este nuevo sistema de tipos ampliado será el de anotar los tipos y demás entidades de un programa Java para describir declarativamente sus propiedades o su comportamiento esperado, para poder comprobar, a través de los procesadores de anotaciones acoplados al compilador, si las reglas que impone la semántica definida por sus anotaciones se cumplen y, en el caso de que no se cumplieran, arrojar el correspondiente warning o error de compilación.

19.2.- Nuevos @Target: TYPE_PARAMETER y TYPE_USE.

La especificación **JSR 308** define **dos nuevos tipos de elementos objetivo** donde pueden ubicarse las anotaciones: las **anotaciones sobre parámetros de tipo** (que se incluyen ahora porque no fueron incluidas por descuido en la especificación JSR 175 de Java SE 5) y las **anotaciones sobre “usos de tipos”**.

Para modelar estos dos nuevos tipos de elementos objetivo se han introducido dos nuevos valores en el tipo enumerado **ElementType** que pueden usarse como valores válidos en la meta-anotación **@Target**:

Meta-Anotación @Target – Novedades Java SE 8 – Nuevos valores permitidos	
Tipo de elemento	Descripción
TYPE_PARAMETER	Tipo anotación aplicable a parámetros de tipo.
TYPE_USE	Tipo anotación aplicable a los “usos de tipos”.

19.3.- Anotaciones sobre parámetros de tipo.

Como hemos visto en el apartado anterior, en la especificación original de las anotaciones para Java SE 5, la JSR 175, se cometió el descuido de no incluir soporte para las anotaciones sobre parámetros de tipo. La especificación JSR 308 ha corregido ese “error histórico” incluyendo por fin soporte para poder anotar parámetros de tipo en cualquiera de sus apariciones, ya sea en definiciones de clase, interfaces o en la definición de métodos.

Ejemplo: Anotaciones sobre parámetros de tipo en clase, interfaz y método

```
public class ClaseEjemplo<@ImprimirMensaje("Parámetro T") T> { ... }
public interface InterfazEjemplo<@ImprimirMensaje("Parámetro T") T> { ... }
public <@ImprimirMensaje("Parámetro T") T> T metodoEjemplo(T item) { ... }
```

Ejemplo: Anot. sobre método que puede parecer sobre parámetro de tipo

CUIDADO: A la hora de anotar los parámetros de tipo de un método, hay que asegurarse de que la anotación se realiza sobre la declaración del parámetro, que es la parte de código que está entre corchetes angulares `< >`. Si la anotación no está dentro de los corchetes angulares, desde el punto de vista del procesamiento de anotaciones, se entenderá que la anotación es sobre el método y no sobre su parámetro de tipo.

El siguiente ejemplo de anotación sería una anotación sobre el método `metodoEjemplo`, y no sobre el tipo parámetro `T`, tal y como aparentemente podría parecer.

```
public @ImprimirMensaje("Anotación sobre el metodoEjemplo") T metodoEjemplo(T item) { ... }
```

19.4.- Anotaciones sobre usos de tipos.

La principal nueva característica de la especificación **JSR 308** (Annotations on Java Types), es la que le da su nombre: las “**anotaciones sobre tipos Java**”. En este apartado vamos a entrar en detalle de lo que implica la introducción de esta nueva característica en la forma de programar en Java y en el día a día de los desarrolladores.

Las “**anotaciones sobre tipos**” son una nueva característica añadida al compilador Java que permite colocar anotaciones, en lugar de donde fuera posible ubicar modificadores como hasta ahora, allí donde sea posible ubicar tipos, por lo que ahora pueden ubicarse anotaciones sobre los nombres de tipos que supongan para el compilador un “uso de tipo”, concepto que estudiaremos a continuación.

Esto permite un mejor análisis del código fuente Java al poder enriquecer nuevos elementos del código con información declarativa acerca de su semántica y comportamiento esperado, lo que permitirá un **procesamiento de anotaciones más flexible y potente**.

19.4.1.- Concepto de “uso de tipo”.

Un concepto clave para entender cómo funcionan las “anotaciones sobre tipos” es el concepto de “**uso de tipo**”, por lo que trataremos de delimitarlo lo más exactamente posible.

El lenguaje Java usa los nombres de tipos de 3 formas distintas:

- 1) Definiciones y Declaraciones de tipos (las cuales ya eran anotables desde Java SE 5).
- 2) Usos de tipos (que son los que ya se pueden anotar desde Java SE 8).
- 3) Referencias de tipos (meta-referencias a los tipos para otras necesidades).

Definición de “uso de tipo”

¿Cómo discernir entonces en el código lo que es un “uso de tipo” y lo que no? JSR 308 da la siguiente **definición** de “uso de tipo”:

Es un “uso de tipo” toda ocurrencia del nombre de un tipo que se asocie en tiempo de ejecución con instancias o variables de dicho tipo.

¿Qué expresiones SÍ son “usos de tipos”?

Según la definición que hemos dado, **serían “usos de tipos” las apariciones de nombres de tipos en las definiciones y declaraciones de tipos, así como en las declaraciones de variables y en signaturas de métodos.**

¿Qué expresiones NO son “usos de tipos”?

NO serían “usos de tipos” los de nombres de tipos en sentencias `import`, ya que en estas últimas la ocurrencia del nombre del tipo no es un uso propiamente dicho del tipo donde variables se vayan a vincular posteriormente con él, si no un mecanismo sintáctico que proporciona cierta meta-information de ámbito para el compilador. **Tampoco son “usos de tipos” los literales de clase, los accesos a miembros estáticos o aquellos usos del nombre de un tipo que representan propiamente una anotación.**

19.4.2- Ejemplos: Anotaciones sobre usos de tipos correctas.

Lo siguiente son ejemplos válidos de uso de las anotaciones sobre tipos para que nos familiaricemos con su sintaxis. La anotación siempre precede al tipo que se está anotando (véase los ejemplos sobre Arrays, donde hay que tener especial cuidado). Los nombres de todos los tipos anotación de los ejemplos son ficticios, sólo son la ilustración de diversas posibilidades interesantes.

Anotaciones sobre usos de tipos – Ejemplos de anotaciones correctas	
Clases e Interfaces	
Herencia	<code>public class UnmodifiableList<T> implements @Readonly List<T> { ... }</code>
Invocación de Constructores	<code>new @NonEmpty @Readonly List<String>(miConjuntoStringsNoVacio);</code>
Inv. Constructores de tipos anidados	<code>miObjeto.new @Readonly ClaseAnidada();</code>
Variables	
Variables	<code>@Email String str1; @NotNull @NotBlank String str2;</code>
Var. tipos anidados	<code>Map.<@NotNull Entry entry;</code>
Castings	<code>miStringNoNulo = (@NotNull String) miObjeto; x = (@A Tipo1 & @B Tipo2) y;</code>
Arrays	<code>// array de arrays de documentos (Document) de sólo lectura @Readonly Document [][] docs1 = new @Readonly Document [2][12]; // array (Document[]) de sólo lectura de arrays de documentos Document @Readonly [][] docs2 = new Document @Readonly [2][12]; // array de arrays (Document[][][]) de sólo lectura de documentos Document[] @Readonly [] docs3 = new Document[2] @Readonly [12];</code>
Genericidad	
Variables con arg. genéricos	<code>List<@Email String> emails; Map<@NotNull String, @NonEmpty List<@Readonly Document>> documentos;</code>
Invocación de método genérico	<code>obj.<@NotNull String>metodo(miStringNoNulo);</code>
Constr. genéricos	<code>new <String> @Interned MiClaseGenerica()</code>
Parámetros de tipo	
Límites en parámetro de tipo	<code>public class Folder<F extends @Existing File> { ... }</code>
Comodín	<code>List<@Immutable ? extends Comparable<T>> listaInmutableYComparable;</code>
Límites en tipos comodín	<code>Collection<? super @Existing File></code>
Tipo intersección	<code>public <T extends @ReadOnly List<T> & @Localized Message> void m(...)</code>
Control de errores	
Excepciones	<code>void monitorTemp() throws @Critical TemperatureException { ... }</code>
Reflexión sobre tipos	
Pruebas sobre tipos	<code>boolean isNotNull = str instanceof @NotNull String; boolean isNotBlankEmail = str instanceof @NotBlank @Email String;</code>
Referencias a ejecutables	
Métodos	<code>@Winter Date::getDay</code>
Métodos genéricos	<code>List<@English String>::size</code>
Constructores	<code>@NonZero RandomNumber::new</code>
Constr. genéricos	<code>HashSet<@Student Person>::new</code>

19.4.3- Ejemplos: Anotaciones sobre usos de tipos incorrectas .

Lo siguiente son ejemplos NO válidos de uso de las anotaciones sobre tipos en los que el compilador nos arrojará un error de compilación. Al igual que en el apartado de ejemplos de anotaciones correctas, los nombres de todos los tipos anotación de estos ejemplos son ficticios.

Anotaciones sobre usos de tipos – Ejemplos de anotaciones incorrectas	
Imports	
Importaciones	<code>import @NotNull java.util.Date; // ERROR</code> <code>import java.util.@NotNull Date; // ERROR</code>
Importaciones estáticas	<code>import static @Inmutable java.lang.Math.PI; // ERROR</code> <code>import static java.lang.@Inmutable Math.PI; // ERROR</code>
Clases e Interfaces	
Literales de Clase	<code>@Even int.class // ERROR</code> <code>int @NonEmpty [].class // ERROR</code>
Literales de Interfaz	<code>@NonEmpty List.class // ERROR</code> <code>@NotNull Stream.class // ERROR</code>
Referencias a la superclase	<code>@NotNull MiClase.super.nombreCampoSuperclase // ERROR</code> <code>@Readonly MiClase.super::nombreMetodoSuperclase // ERROR</code>
Variables	
Variables estáticas	<code>valorPI = @NotNull java.lang.Math.PI; // ERROR</code>
Clases anidadas	
Referencia a campos estáticos de las clases anidadas	<code>@ReadOnly ClaseExterior.campoAnidadoEstatico // ERROR</code>
Referencia a las clases estáticas anidadas (excepto la última)	<code>@ReadOnly ClaseExterior.ClaseAnidadaEstatica // ERROR</code>
Referencia a la última clase estática anidada	<code>ClaseExterior.@ReadOnly ClaseAnidadaEstatica // OK</code> <code>(véase explicación más abajo)</code>

Recordemos que no podemos anotar los “usos sintácticos de tipos”, que realmente no son “usos de tipos” (relacionados con una instancia del tipo). Por tanto, las ocurrencias de tipos que aporten meta-information al compilador, o que hagan referencia a miembros estáticos no serán anotables (los miembros estáticos se refieren a los tipos y no a instancias de dichos tipos).

En las cadenas de nombres de clases anidadas sólo el último es “uso de tipo”

En la tabla hemos visto que dos expresiones aparentemente muy parecidas daban resultados en el compilador totalmente diferentes:

```
@ReadOnly ClaseExterior.ClaseAnidadaEstatica // ERROR
ClaseExterior.@ReadOnly ClaseAnidadaEstatica // OK
```

Para entender esto, vamos a reflexionar un momento sobre el concepto de “nombre de tipo como mecanismo de ámbito”. El nombre de `ClaseExterior` está funcionando simplemente aquí como un mecanismo de ámbito para ubicar a la `ClaseAnidadaEstatica`. Es el mismo papel que hace el nombre del paquete “`java.lang`” en un nombre de clase como “`java.lang.Object`”.

En cambio, el último nombre de la “cadena” ya no es una referencia necesaria para determinar el ámbito de la clase, si no que identifica una clase real de la que se quiere crear una instancia. Así pues, es un “uso de tipo” correcto y se puede anotar correctamente:

```
ClaseExterior.@ReadOnly ClaseAnidadaEstatica x = ...; // OK :-)
```

19.4.4- Anotaciones multi-target con múltiples elementos afectados.

Previamente a JSR 308, al no poder anotarse los usos de tipos, los elementos de código fuente que se podían anotar eran menos y estaban más aislados.

No obstante, usando JSR 308, al poder anotar usos de tipos, **puede ocurrir que una única anotación afecte a más de un elemento de código al mismo tiempo**, siempre y cuando su tipo anotación sea “multi-target”, esto es, que pueda aplicarse a varios elementos de código al estar definido para anotar múltiples @Target.

En principio, dado el escaso alcance sintáctico de una anotación (ya que sólo afecta a su elemento más inmediatamente próximo hacia la derecha que no sea a su vez otra anotación), este caso tan especial puede darse en anotaciones que anoten declaraciones de variables en los que el tipo anotación pueda entenderse aplicado al mismo tiempo sobre el uso del tipo de la declaración y sobre la variable al mismo tiempo.

Ejemplo: Anotación que afecta a un uso de tipo y a una variable local

Imaginemos que el tipo anotación ficticio `@NotNull` es multi-target, aplicándose, entre otros, a las variables locales, así como a los usos de tipo. En ese caso, dicho tipo anotación estará definido usando estos tipos de target como: `@Target({ElementType.LOCAL_VARIABLE, ElementType.TYPE_USE, ...})`. Siendo así, fijémonos en la siguiente declaración de variable local anotada con `@NotNull`:

```
@NotNull String vrbleCadena = "abc"; // afecta a String y a vrbleCadena al mismo tiempo
```

Puesto que `@NotNull` es aplicable al tipo `String` y aplicable a la variable local `vrbleCadena`, en este caso la anotación se entendería aplicada a ambos elementos.

Lógicamente, esta duplicidad no ocurriría si `@NotNull` no fuera multi-target y se aplicara sólo a variables locales o sólo a usos de tipo.

Nótese que esto es así porque la sintaxis de Java no permite ubicar una segunda anotación justo antes del nombre de la variable, ya que esto se entendería como el tipo de la variable y el compilador arrojaría un error de tipo no encontrado:

```
@NotNull String @NotNull vrbleCadena = "abc"; // ERROR: el tipo String.NotNull no existe
```

Ejemplo: Anotación que afecta a un uso de tipo y a un parámetro formal de método

Si suponemos que el tipo anotación ficticio multi-target `@NotNull` del ejemplo anterior también es aplicable, además de a variables locales y usos de tipo, a los parámetros formales, tendríamos: `@Target({ElementType.LOCAL_VARIABLE, ElementType.TYPE_USE, ElementType.PARAMETER, ...})`.

Siendo así, ocurrirá por tanto que las anotaciones aplicadas sobre parámetros formales, ya sea en la declaración de métodos o de constructores, afectarán también a ambos, tipo y parámetro:

```
public void metodo(@NotNull String vrbleCadena) // afecta a String y a vrbleCadena
public ClaseAContruir(@NotNull String vrbleCadena) // afecta a String y a vrbleCadena
```

19.4.5- Sintaxis de tipo receptor explícito.

Necesidad de un receptor

Hay una construcción sintáctica que, debido a sus especiales características, ha de tratarse de forma un tanto especial para poder realizar anotaciones de usos de tipos sobre ella. Por ejemplo, imaginemos que tenemos una clase **FicheroBinario** como la siguiente:

```
public class FicheroBinario {  
  
    public boolean abrirFichero() throws IOException { ... }  
    public byte[] leerFichero() throws IOException { ... }  
}
```

Para invocar dicho método, suponiendo que la clase dispone de un constructor en el que se le pasa la ruta física al fichero, tendríamos que escribir algo similar a lo siguiente:

```
FicheroBinario fichero = new FicheroBinario("C:\\fichero.bin");  
byte[] bytesLeidos = fichero.leerFichero();
```

Si ahora quisiéramos anotar el método **leerFichero** para que se invocara únicamente sobre ficheros previamente abiertos con el tipo anotación ficticio **@Open** y poder así detectar y prevenir futuros errores en tiempo de ejecución, teniendo en cuenta que el método no tiene argumentos, ¿cómo podríamos anotar el objeto sobre el que se invoca el método si las anotaciones de usos de tipo requieren de un nombre de tipo?

```
public byte[] leerFichero(@Open) throws IOException { ... } // ERROR DE COMPILACIÓN
```

Si intentamos ubicar una anotación directamente en la lista de parámetros formales del método, el compilador nos arroja el siguiente error:

```
Syntax error, insert "Type VariableDeclaratorId" to complete FormalParameterList
```

Ello se debe a que la anotación no está siendo aplicada sobre alguno de los parámetros formales del método, tal y como espera el compilador. Para solucionar esta restricción sintáctica, es necesario introducir el concepto de **receptor**.

Concepto de receptor

Se llama **receptor** al parámetro formal implícito **this** que tienen todos los métodos de instancia (no estáticos) de una clase. No está permitido para métodos estáticos o expresiones lambda. Así pues, para los métodos de instancia, se puede declarar explícitamente el receptor del método como primer parámetro formal del mismo de la siguiente manera:

```
public byte[] leerFichero(FicheroBinario this) throws IOException { ... }
```

Como vemos, se ha incluido un parámetro formal que tiene como tipo la clase actual en la que el método está siendo definido y como nombre **this**. Tiene que llamarse forzosamente **this** o, si no, el compilador no lo interpretará como el parámetro formal implícito del tipo receptor, si no como otro parámetro formal más.

Anotación sobre el uso de un tipo receptor

Teniendo nuestro parámetro formal implícito de tipo receptor escrito explícitamente, ya podemos anotar dicha ocurrencia del tipo con las anotaciones sobre dicho uso de tipo que sean necesarias simplemente anteponiéndolas al nombre del tipo receptor del parámetro implícito:

```
public byte[] leerFichero(@Open FicheroBinario this) throws IOException { ... } // OK
```

IMPORTANTE: La información sobre las anotaciones de tipos colocadas sobre el tipo receptor implícito se guardan en el fichero de clase generado por el compilador, en la estructura correspondiente al método, llamada “method_info”, dentro de su atributo “*TypeAnnotations”. Esto quiere decir que, en el caso del ejemplo, a la hora de recuperarlas, se recuperarán como anotaciones de uso de tipo sobre el método `leerFichero` cuyo tipo anotado deberá ser el tipo receptor implícito `FicheroBinario`.

Para poder anotar un tipo receptor, el tipo anotación deberá tener como uno de los valores de su meta-anotación `@Target` el tipo de elemento de las anotaciones sobre usos de tipos, `ElementType.TYPE_USE`, o el compilador dará error.

NOTA: Esto es así porque las anotaciones sobre los demás parámetros formales se considerarán por defecto anotaciones sobre declaraciones (o sobre tipos y declaraciones al mismo tiempo, como se ha explicado en el apartado anterior). Sin embargo, una anotación sobre el primer parámetro formal implícito, es decir, una anotación sobre el tipo receptor implícito, sólo puede considerarse por parte del compilador como anotación de tipo, ya que la especificación explícita del parámetro receptor receptor no puede entenderse como una declaración, si no como un mecanismo sintáctico especial.

Uso práctico de las anotaciones sobre un tipo receptor

Con la anotación `@Open` ya colocada como queríamos, si escribiésemos un procesador de anotaciones para la misma que comprobara si el objeto de tipo `FicheroBinario` sobre el que se invoca el método está efectivamente abierto y, en caso contrario, arrojara error de compilación, el código anterior provocaría error de compilación al invocar el método `leerFichero` sobre un fichero que no ha sido abierto previamente:

```
FicheroBinario fichero = new FicheroBinario("C:\\fichero.bin");
byte[] bytesLeidos = fichero.leerFichero(); // ERROR DE COMPILACIÓN (fichero no abierto)
```

Sin embargo, si abrimos el fichero antes de leerlo, el compilador no se quejará:

```
FicheroBinario fichero = new FicheroBinario("C:\\fichero.bin");
boolean ficheroAbierto = fichero.abrirFichero(); // abrimos el fichero antes de leerlo
byte[] bytesLeidos = fichero.leerFichero(); // OK
```

De esta forma, hemos aplicado el principio general que hay detrás de las anotaciones de tipos de mejorar el análisis y la calidad del código, y descubrir el mayor número posible de errores en tiempo de compilación para hacer el código más robusto y eficiente.

Tratamiento transparente del parámetro receptor implícito

El parámetro formal del tipo receptor, que como ya sabemos es opcional, no tiene ningún efecto sobre el código generado por el compilador para los ficheros de clase, ni tampoco tiene ningún efecto sobre la ejecución de dicho código. Sólo sirve únicamente como lugar de anclaje donde colocar las anotaciones sobre el uso del tipo receptor.

Además, en cuanto al método en sí, el compilador generará exactamente el mismo *bytecode* y la API de reflexión devolverá los mismos resultados acerca del número de parámetros formales del método, esté presente o no de forma explícita el receptor. El parámetro formal del tipo receptor no se traslada como un parámetro formal real en el código finalmente compilado. Es una convención sintáctica especial que es como si no existiera a la hora de la verdad cuando se considera la definición de la signatura de un método. Por tanto, al considerar la signatura de un método, hemos de ignorar la presencia explícita de un parámetro formal para el tipo receptor, ya que es transparente en lo que respecta a su signatura definitiva.

Receptor en métodos de clases anidadas

Las clases anidadas pueden tener no ya uno, si no múltiples receptores, en función del nivel de anidamiento propiamente dicho de la clase anidada en cuestión. Por ejemplo, dada la siguiente definición de clases anidadadas:

```
public class Exterior {
    class Media {
        class Interior {
            void metodoClaseInterior(@A Exterior.@B Media.@C Interior this) { ... }
        }
    }
}
```

El método `metodoClaseInterior` tiene 3 receptores, uno por cada clase:

- Para la clase `Interior`: `this`
- Para la clase `Media`: `Media.this`
- Para la clase `Exterior`: `Exterior.this`

Además, como podemos observar, el uso de cada uno de los múltiples tipos receptor puede ser anotado independientemente con las anotaciones sobre usos de tipos que sean necesarias.

Receptor en constructores de clases anidadas

Los constructores, a diferencia de los métodos, no tienen receptores, salvo en el caso de que sean constructores de clases anidadas. Por ejemplo, para una definición de constructor de una clase anidada, se incluiría el parámetro formal del tipo receptor de la siguiente manera:

```
public class Exterior {
    class Interior {
        // constructor
        @Resultado Interior(@Receptor Exterior Exterior.this) { ... }
    }
}
```

Nótese cómo en el caso de los receptores de los constructores, el parámetro receptor va cualificado por el nombre de la clase anidada, `Exterior.this`, mientras que esto no era así en el caso de los métodos, donde sólo se escribía `this` exclusivamente.

Esto es así porque en los constructores, según la especificación del lenguaje Java, **this** hace referencia al resultado del método constructor. Así pues, existe una diferencia de criterio a tener en cuenta, ya que en los métodos **this** es una referencia al receptor y en los constructores **this** es una referencia al objeto resultado de la construcción. Por este motivo, para referenciar a los receptores de los constructores de las clases anidadas se utiliza esta excepción a la sintaxis habitual que es cualificar el receptor con el nombre de la clase padre: **Exterior.this**.

Receptor en clases anónimas

Una clase anónima no permite anotaciones sobre el uso de su tipo debido precisamente a que la clase no tiene un nombre con el cual poder referenciar el tipo receptor. Por tanto, no es posible anotar los tipos receptor de sus métodos o constructores. Si dicha funcionalidad fuera necesaria, lo adecuado sería redefinir la clase como clase nominal con un nombre propio.

19.5.- Carencias del procesamiento hasta Java SE 8.

Debido a defectos de diseño de la implementación inicial del formato interno de los ficheros de clase Java, el procesamiento de anotaciones exhibió desde el principio algunas graves carencias, como ya indicamos en los apartados introductorios del presente manual. Afortunadamente, dichas carencias fueron resueltas en Java SE 8 gracias a la implementación ampliada descrita por la especificación JSR 308.

No obstante, el que dichas carencias hayan sido subsanadas en Java SE 8 no nos exime de analizarlas en este manual, ya que, además del interés docente del tema en sí, estas carencias han estado presentes hasta Java SE 7. Finalmente, es recomendable conocerlas para saber por qué fue necesario introducir los cambios descritos por la especificación JSR 308.

El principal inconveniente de las anotaciones tal y como se diseñaron inicialmente para J2SE 1.5 por parte de Sun Microsystems fue que el procesamiento de anotaciones no estaba dentro del compilador, sino que se realizaba a través de **apt**. Esto fue lo primero que se corrigió en Java SE 6 con **JSR 269** (Pluggable Annotation Processing API), aprovechando además para redefinir la API de procesamiento propiamente dicha (**javax.annotation.processing**) e introducir una nueva API que modelaba los elementos del lenguaje Java (**javax.lang.model**).

Con la especificación **JSR 269** se le dio un auténtico lavado de cara al procesamiento de anotaciones para hacerlo más accesible, extensible y universal (detalle importante este último, ya que **apt** no formaba parte de la plataforma Java, si no sólo de la implementación de Sun).

No obstante, incluso tras la inclusión de la JSR 269, hubo dos **carencias graves** que no fueron corregidas y de las que a continuación nos vamos a ocupar en mayor detalle:

- **La información incompleta en el formato de fichero de clase (.class).**
- **La imposibilidad del procesamiento de anotaciones sobre variables locales.**
- **La imposibilidad del procesamiento de parámetros de tipo.**

Estas carencias están relacionadas entre sí. A continuación, dedicaremos un apartado específico para examinar en mayor detalle cada una de ellas, para así poder comprender la naturaleza del origen de las mismas y su efecto sobre el procesamiento de anotaciones.

19.5.1.- Información incompleta en el fichero de clase.

Como se dice en la especificación [JSR 308](#) (Annotations on Java Types), en su apartado “[Class file format extensions](#)” (“Extensiones del formato de ficheros de clase”), la información acerca de las anotaciones se guardan en los ficheros de clase (.class) dentro de unas estructuras llamadas *atributos*.

Un *atributo* asocia diversos datos con un elemento de programa, por ejemplo: los *bytecodes* de un método se almacenan en el atributo **Code** de dicho método.

El atributo **RuntimeVisibleAnnotations** almacena la información de las anotaciones de la clase que son accesibles en tiempo de ejecución usando reflexión (política de retención RUNTIME) y el atributo **RuntimeInvisibleAnnotations** almacena las anotaciones de la clase que no son accesibles en tiempo de ejecución usando reflexión (política de retención CLASS). Esta misma jerga aparece cuando utilizamos APIs de manipulación de ficheros de clase como puede ser [Javassist](#).

Los atributos de las anotaciones contienen arrays de estructuras **annotation**, que a su vez contienen arrays de pares **element_value**. Estos pares almacenan los nombres y los valores de los argumentos de una anotación.

Por ejemplo, para una anotación con política de retención RUNTIME como @Anotacion(clave1=valor1, clave2=valor2) se crearía pues una entrada en el atributo **RuntimeVisibleAnnotations** que sería de tipo estructura **annotation** y que tendría dos entradas en su array de pares **element_value** con los valores (clave1, valor1) y (clave2, valor2).

Anotaciones sobre diversos elementos de código se almacenan en diversas estructuras:

- Anotaciones sobre campos: como atributos de la estructura **field_info** del campo.
- Anotaciones sobre métodos: como atributos de la estructura **method_info** del método.
- Anotaciones sobre clases: como atributos de la estructura **attributes** de la clase.

Pero, como se puede observar, ¡¡no se definió un atributo o estructura donde poder guardar la información acerca de las anotaciones sobre variables locales!!!

JSR 308 introduce dos nuevos tipos de atributos: **RuntimeVisibleTypeAnnotations** y **RuntimeInvisibleTypeAnnotations** que, en lugar de contener elementos de tipo estructura **annotation**, define un nuevo tipo estructura **type_annotation** (apartado de la especificación 3.1 The **type_annotation** structure) con información extendida sobre las anotaciones.

La estructura **type_annotation** tiene a su vez una estructura llamada **target_info** (apartado 3.3 The **target_info** field: identifying a program element) que permite identificar de forma única el elemento objetivo sobre el que aplica una anotación concreta. De esta forma, se viene a completar la información que faltaba en el formato de fichero de clase acerca de las anotaciones y así poder permitir el procesamiento de anotaciones sin límites de ninguna clase.

19.5.2.- Procesamiento de anotaciones sobre variables locales.

Como acabamos de ver en el apartado anterior, la especificación **JSR 308** completó la información que faltaba acerca de las anotaciones en el fichero de clase gracias al nuevo tipo de estructura **type_annotation** (apartado de la especificación 3.1 The **type_annotation** structure).

Siendo así, ¿cómo almacena este nuevo tipo de estructura la información que necesitamos para poder realizar el procesamiento de anotaciones sobre variables locales?

El quid de la cuestión es que la estructura **type_annotation** tiene a su vez una estructura llamada **target_info** (apartado de la especificación 3.3 The **target_info** field: identifying a program element) que permite identificar de forma única el elemento de código objetivo sobre el que se aplica una anotación concreta.

Según el valor del tipo de elemento objetivo, dado por la propiedad **target_type** (apartado de la especificación 3.2 The **target_type** field: the type of annotated element), la estructura **target_info** puede constar de un conjunto totalmente diferente de campos. En el apartado de la especificación 3.3.7 Local variables and resource variables, se define la información que se guarda en una estructura **target_info** para el caso de una variable local:

```
localvar_target {
    u2 table_length;
{
    u2 start_pc;
    u2 length;
    u2 index;
} table[table_length];
};
```

Esta información se añadirá a un atributo **Runtime[In]visibleTypeAnnotations** (visible o invisible según la política de retención definida para el tipo anotación en cuestión), que contendrá un **localvar_target** en la tabla de atributos del atributo **Code** correspondiente a la variable local objetivo sobre la que se esté aplicando la anotación.

Con el añadido de esta información a los ficheros de clase, por fin es posible acceder a la información de las anotaciones aplicadas sobre variables locales, tanto por parte de las herramientas de manipulación de ficheros de clase, como por parte de las propias APIs de reflexión del lenguaje Java, solventando por tanto finalmente esta grave carencia que suponía no poder procesar anotaciones aplicadas sobre variables locales.

19.5.3.- Procesamiento de anotaciones sobre parámetros de tipo.

No hay que confundir la anotación de parámetros de tipo con la anotación de “usos de tipos”, aunque ambos hayan sido introducidos en la especificación **JSR 308**. Las anotaciones sobre parámetros de tipo son anotaciones sobre declaraciones, no sobre tipos. El problema fue que no se incluyeron en la especificación original de las anotaciones **JSR 175** por un simple descuido, ya que no se consideraron importantes. La especificación **JSR 308** ha aprovechado la oportunidad de enmendar ese error incluyéndolas, pero no deben confundirse ambos tipos de anotaciones, puesto que son diferentes, aunque hayan sido incluidas al mismo tiempo.

20.- PROCESAMIENTO EN JAVA SE 8.

20.1.- Introducción.

Esencialmente, el procesamiento de anotaciones en Java SE 8 es idéntico al realizado en Java SE 6/7 a nivel estructural. El único factor adicional a tener en cuenta es el tratamiento de las nuevas anotaciones sobre tipos, para lo cual es necesario conocer las novedades en las APIs ya vistas.

20.2.- Problema: Descubrimiento de anotaciones sobre tipos.

A pesar de lo muy publicitado que fue por parte de Oracle la nueva funcionalidad de las anotaciones sobre usos de tipos, a la hora de la verdad, cuando nos disponemos a procesarlas, nos encontramos con una desagradable sorpresa: **las anotaciones sobre usos de tipos no están incluidas en el proceso de descubrimiento de anotaciones**.

Esto significa que ninguno de los métodos utilizados para el descubrimiento de anotaciones de la clase `RoundEnvironment` devolverá instancias de elementos anotados con anotaciones sobre tipos. Nuestras pruebas con la API actualizada de Java SE 8 así lo indicaban y esto quedó confirmado con las declaraciones que Joseph Darcy, ingeniero jefe de procesamiento de anotaciones en Oracle, hizo en el grupo de correo [type-annotations-spec-experts](#) en el mensaje de 1 de Abril de 2013 titulado «[FYI, JSR 308 and the annotation processing discovery process](#)», en el que comentaba lo siguiente (se ha traducido el texto del original en inglés):

[...] Si una anotación aparece sobre una declaración, se incluirá en el [proceso de] descubrimiento y si una anotación aparece sobre un uso de tipo, *no* se incluirá en el [proceso de] descubrimiento. Si un tipo anotación es declarado como capaz de aplicarse a ambos, declaraciones y usos de tipos, sólo las primeras aplicaciones serán visibles para el [proceso de] descubrimiento.

[...] La API de procesamiento de alrededor de 2006 – 2006 se construyó sobre la suposición de que las anotaciones serían sólo sobre declaraciones / elementos. Por ejemplo, para una ronda de procesamiento hay método que retornan las declaraciones / elementos anotados con un tipo anotación particular:

```
RoundEnvironment.getElementsAnnotatedWith(Class class)
RoundEnvironment.getElementsAnnotatedWith(TypeElement typeElement)
```

Propiedades de consistencia como “dado el conjunto de anotaciones a ser procesadas en una ronda, puedes encontrar todos los constructos anotados con esas anotaciones iterando sobre los elementos anotados con cada anotación” no podrían mantenerse si las anotaciones sobre usos de tipos se incluyeran en el [proceso de] descubrimiento.

Esta clase de razonamiento es útil, pero generalmente no es la clase de material que ubicamos en el javadoc de una API. Sin embargo, ciertamente será pertinente para la revisión de mantenimiento de JSR 269 para Java SE 8.

IMPORTANTE: Esto supone un grave inconveniente para los desarrolladores de procesadores de anotaciones, ya que **será necesario descubrir las anotaciones sobre usos de tipos de forma manual** mediante otras estrategias menos directas y más rebuscadas.

En el apartado “Resumen de resultados del proceso de descubrimiento” del procesamiento JSR 269 expusimos una tabla aclaratoria de cómo se comportaban las APIs JSR 269 y la Tree API a la hora de descubrir y procesar anotaciones. Aquí presentamos dicha tabla ampliada y actualizada a Java SE 8:

Resultados del proceso de descubrimiento de anotaciones según su ubicación *** para Java SE 8 ***						
Ubicación de la anotación	javac	JSR 269		Compiler Tree API		
	¿Compila? (Java SE 8)	¿Elem. modelado? (ElementKind)	¿Elemento descubrible?	¿Árbol modelado? (Tree.Kind)	¿Árbol descubrible?	¿Convertible en elemento?
Paquete	SÍ	PACKAGE	SÍ	COMPILATION_UNIT	SÍ (manual)	SÍ
Clase	SÍ	CLASS	SÍ	CLASS	SÍ (manual)	SÍ
Enumerado	SÍ	ENUM	SÍ	ENUM	SÍ (manual)	SÍ
Interfaz	SÍ	INTERFACE	SÍ	INTERFACE	SÍ (manual)	SÍ
Tipo anotación	SÍ	ANNOTATION_TYPE	SÍ	ANNOTATION_TYPE	SÍ (manual)	SÍ
Constructor	SÍ	CONSTRUCTOR	SÍ	METHOD	SÍ (manual)	SÍ
Método	SÍ	METHOD	SÍ	METHOD	SÍ (manual)	SÍ
Campo	SÍ	FIELD	SÍ	VARIABLE	SÍ (manual)	SÍ
Constante de un tipo enumerado	SÍ	ENUM_CONSTANT	SÍ	VARIABLE	SÍ (manual)	SÍ
Parámetro	SÍ	PARAMETER	SÍ	VARIABLE	SÍ (manual)	SÍ
Variable local	SÍ	LOCAL_VARIABLE	NO	VARIABLE	SÍ (manual)	SÍ
Parám. de excepción	SÍ	EXCEPTION_PARAMETER	NO	VARIABLE	SÍ (manual)	SÍ
Variable de recurso	SÍ	RESOURCE_VARIABLE	NO	VARIABLE	SÍ (manual)	SÍ
Parámetro de tipo	SÍ (nuevo)	TYPE_PARAMETER	SÍ (nuevo)	TYPE_PARAMETER	SÍ (manual)	SÍ (nuevo)
Inicializador estático	NO	STATIC_INIT	NO	BLOCK	SÍ (manual)	NO
Inicializador de instancia	NO	INSTANCE_INIT	NO	BLOCK	SÍ (manual)	NO

Resultados del proceso de descubrimiento de anotaciones sobre usos de tipos según su ubicación						
Uso de tipo sobre...	javac	JSR 269		Compiler Tree API		
	¿Compila? (Java SE 8)	¿Tipo modelado? (TypeKind)	¿Tipo descubrible?	¿Árbol modelado? (Tree.Kind)	¿Árbol descubrible?	¿Convertible en elemento?
Array	SÍ (nuevo)	ARRAY	NO	ARRAY_TYPE	SÍ (manual)	NO
Clase/Interfaz	SÍ (nuevo)	DECLARED	NO	ANNOTATED_TYPE	SÍ (manual)	NO
Tipo parametrizado	SÍ (nuevo)	DECLARED	NO	PARAMETERIZED_TYPE	SÍ (manual)	SÍ
Tipo primitivo	SÍ (nuevo)	BOOLEAN, BYTE, CHAR, DOUBLE, FLOAT, INT, LONG, SHORT	NO	PRIMITIVE_TYPE	SÍ (manual)	NO
Tipo intersección	SÍ (nuevo)	INTERSECTION	NO	INTERSECTION_TYPE	NO (se busca sólo en castings)	NO
Tipo unión	SÍ (nuevo)	UNION	NO	UNION_TYPE	SÍ (manual)	NO

IMPORTANTE: Como vemos, las APIs de procesamiento de anotaciones no son de ninguna ayuda a la hora de descubrir anotaciones sobre usos de tipos. Y en el caso de la Compiler Tree API, nos permite descubrir la mayoría de las estructuras sobre las que se aplican las anotaciones de usos de tipos, pero los objetos que nos ofrece no se pueden convertir en elementos de la Language Model API y tampoco podemos recuperar de ellos información alguna sobre anotaciones de tipos. Para poder recuperar información sobre las anotaciones sobre usos de tipos habremos de descansar fuertemente sobre la API de reflexión del lenguaje, como veremos en el siguiente apartado.

20.3.- Uso de las diferentes APIs para el procesamiento.

Como hemos introducido en el apartado anterior, las APIs estándar de procesamiento de anotaciones no resuelven correctamente el proceso de descubrimiento de las nuevas anotaciones sobre usos de tipos. Así las cosas, en este apartado vamos a explicar cómo se deben usar las APIs estándar para poder descubrir las anotaciones que queremos procesar.

Esto plantea un problema grave: para descubrir los diferentes tipos de anotaciones a procesar no quedará otro remedio que usar varias APIs diferentes. A continuación vamos a exponer qué APIs o estrategias será necesario utilizar según el tipo de anotaciones que queramos descubrir. Empezamos con las anotaciones sobre declaraciones:

APIs y estrategias de descubrimiento de anotaciones sobre declaraciones	
Tipos de anotaciones	Método o estrategia de descubrimiento
Anotaciones sobre declaraciones fuera de los bloques de código	<code>RoundEnvironment.getElementsAnnotatedWith(...)</code>
Anotaciones sobre declaraciones dentro de los bloques de código y concretamente anotaciones sobre variables locales	Uso del visitor <code>TreePathScanner</code> para visitar las variables de un tipo con <code>visitVariable(VariableTree node, P p)</code> obteniendo una instancia de <code>VariableElement</code> desde su nodo.

En cuanto al procesamiento de anotaciones sobre usos de tipos, es necesario contemplar los 16 “contextos de tipos” donde puede aparecer un uso de tipo según la especificación del lenguaje Java (Java Language Specification o JLS) apartado 4.11 y el documento de desarrollo de la especificación JSR 308 de anotaciones sobre tipos. Estos contextos son los siguientes:

En declaraciones:

- 1. Un tipo en la cláusula extends o implements de una declaración de clase.
- 2. Un tipo en cláusula extends de una declaración de interfaz.
- 3. El tipo de retorno de un método (incluyendo el tipo de un elemento de un tipo anotación).
- 4. Un tipo en la cláusula throws de un método o constructor.
- 5. Un tipo en la cláusula extends de una declaración de parámetro de tipo de una clase genérica, interfaz, método o constructor.
- 6. El tipo en una declaración de campo de una clase o interfaz (incluyendo constantes de enum).
- 7. El tipo en una declaración de parámetro formal de un método, constructor o expresión lambda.
- 8. El tipo del parámetro receptor de un método.
- 9. El tipo de una declaración de una variable local.
- 10. El tipo de una declaración de un parámetro de excepción en una cláusula catch.

En expresiones:

- 11. Un tipo en la lista explícita de tipos argumento de una invocación al constructor explícito, o una expresión de creación de instancia de una clase, o una expresión de invocación de método.
- 12. En una expresión de creación de instancia de una clase no cualificada, como el tipo de la clase a ser instanciada o como superclase directa o superinteraz directa de una clase anónima a instanciar.
- 13. El tipo elemento en una expresión de creación de un array.
- 14. El tipo en el operador de casting de una expresión de casting.
- 15. El tipo que sigue al operador relacionar instanceof.
- 16. En una expresión de referencia a un método, como el tipo referencia para buscar un método miembro, o como tipo de la clase, o como tipo array a construir.

Conociendo los “contextos de tipos” donde pueden aparecer las anotaciones sobre usos de tipos, en el siguiente cuadro presentamos los métodos o estrategias de descubrimientos para los mismos en el siguiente cuadro:

APIs y estrategias de descubrimiento de anotaciones sobre usos de tipos	
Tipos de anotaciones	Método o estrategia de descubrimiento
Anotaciones sobre usos de tipos de los contextos número 1 a 9 (en declaraciones), y 11-12 (incluidas expresiones lambda)	Recuperar una instancia de elemento de la Language Model API y, a partir de ella, obtener su instancia correspondiente de la API de Reflexión y desde ella obtener instancias de AnnotatedType . <u>NOTA:</u> En el caso del contexto nº 7 no se soportan las expresiones lambda.
Anotaciones sobre usos de tipos de los contextos número 10 y 13-16 (en expresiones)	No existe forma de acceder mediante las APIs estándar a la información de las anotaciones sobre expresiones que no están modeladas como elementos de programa. La Compiler Tree API permite recorrer los árboles sintácticos de estas expresiones, pero <u>no</u> nos ofrece información alguna sobre anotaciones sobre tipos.

20.4.- Ejemplos de código.

Los ejemplos de código para Java SE 8 se encuentran en el código fuente adicional en el proyecto **anotaciones-proc-java8** en el paquete **anotaciones.procesamiento.java8.ejemplos**. Son los ejemplos ya vistos para el procesamiento en Java SE 5 y Java SE 6/7, pero adaptados para funcionar con Java SE 8.

A continuación nos centraremos en el ejemplo del tipo anotación **ImprimirMensaje**, ya que las novedades del procesamiento de anotaciones en Java SE 8 se concentran en las nuevas anotaciones sobre usos de tipos y el proceso de su descubrimiento, cuyos problemas hemos explicado en los apartados anteriores.

Se ha optado en la medida de lo posible por refactorizar los métodos y estrategias de descubrimiento de anotaciones y se han ubicado en el paquete de clases de utilidad del ejemplo del tipo anotación **ImpimirMensaje** (en **anotaciones.procesamiento.java8.ejemplos.imprimirMensaje.utils**). Estas clases de utilidad son las siguientes:

Clases de utilidad para el descubrimiento de anotaciones	
Clase de utilidad	Descripción
BuscadorAnotacionesSobreDeclaraciones	Búsqueda de anotaciones sobre declaraciones. Utiliza la clase BuscadorElementosVariablesLocales .
BuscadorAnotacionesSobreUsosTipos	Búsqueda de anotaciones sobre usos de tipos. Utiliza la clase BuscadorElementosVariablesLocales y la clase AdaptadorAPIReflexion .
BuscadorElementosVariablesLocales	Búsqueda de anotaciones sobre variables locales utilizando un TreePathScanner de Compiler Tree API.
AdaptadorAPIReflexion	Clase auxiliar que permite obtener instancias de la API de Reflexión a partir de instancias de varios tipos de la Language Model API. Los objetos de la API de reflexión permiten recuperar las anotaciones sobre usos de tipos.

21.- CHECKER FRAMEWORK.

21.1.- Introducción.

Como hemos visto, la especificación [JSR 308](#) permite mejorar las capacidades de análisis del código fuente y la comprobación de tipos acoplando módulos al compilador en forma de procesadores de anotaciones JSR 269 que ahora pueden procesar anotaciones ubicadas en nuevos objetivos: en los parámetros de tipo y en los usos de tipo. De esta forma se permite un uso nuevo y muy interesante del procesamiento de anotaciones: la comprobación extendida y avanzada de errores o warnings de compilación a los que no llega el compilador Java estándar a través de los siguientes pasos:

- 1) Creación de un nuevo sistema de tipos más rico en información.
- 2) Anotación del código con el nuevo sistema de tipos.
- 3) Detección de más errores en tiempo de compilación gracias al nuevo sistema de tipos.

De lo que se trata por tanto es de detectar el mayor número posible de errores en el código fuente en tiempo de compilación antes que provoquen problemas en tiempo de ejecución y así además ahorrarse el tiempo necesario para depurar dichos errores, mejorando al mismo tiempo la calidad y la documentación del código gracias a la información que aporta el nuevo sistema de tipos.

No obstante, la distribución de la plataforma Java SE 8 no incluye ninguna API estándar que proporcione dichas funcionalidades de análisis y comprobación de tipos en el código fuente. Para este propósito, se creó en 2006 la especificación [JSR 305](#) (Annotations for Software Defect Detection) para definir una serie de anotaciones estandarizadas que pudieran utilizarse de forma general para la detección de defectos en el código, pero actualmente esta especificación está abandonada. Esto significa que los desarrolladores deben crear sus propias librerías para este propósito, o bien utilizar las librerías desarrolladas por otros.

Checker Framework [<http://checkerframework.org>] es una librería desarrollada por los autores de la especificación JSR 308, pero que no pretende estar oficialmente vinculada ni a dicha especificación ni a Oracle. **Se compone de procesadores de anotaciones llamados “checkers” (“comprobadores”),** que pueden realizar análisis semánticos y otras comprobaciones que **nos permiten detectar errores en nuestro código fuente de forma temprana en tiempo de compilación.**



Checker Framework incluye numerosos “checkers” ya implementados que permiten comprobar la presencia de punteros nulos, mal uso de los bloqueos de concurrencia, fallos o vulnerabilidades de seguridad, sintaxis erróneas de formato o expresiones regulares incorrectas, errores en la conversión de unidades, y otros muchos errores. Además, si ninguno de los “checkers” ofrecidos se ajusta a las necesidades de los usuarios de la librería, esta **ofrece la posibilidad de implementar nuestros propios “checkers” personalizados**, ya sea heredando de clases abstractas básicas para ahorrar esfuerzo de implementación, o incluso desde cero.

Checker Framework tiene un diseño flexible y puede usarse con diversos tipos de herramientas, como entornos de desarrollo (Eclipse o IntelliJ IDEA, entre otros), sistemas de build (como Maven, Gradle, etc) o directamente a través de la línea de comandos. En su página web oficial están las instrucciones para instalar y configurar el framework. En el caso del entorno Eclipse, existe un plugin específico que se integra en la interfaz del entorno para una mayor comodidad de uso.

21.2.- Funcionamiento.

Checker Framework **funciona añadiendo funcionalidad adicional de comprobación semántica** al compilador de Java a través de las posibilidades de las especificaciones dadas por JSR 269 y JSR 308. Esto se lleva a cabo mediante un conjunto de “checkers”, esto es, los procesadores de anotaciones propios de este framework. Cada uno de estos “checkers” o “comprobadores” comprueba y previene una serie de errores semánticos en el código fuente que escapan al control semántico del compilador `javac` estándar.

Checker Framework es **compatible con versiones anteriores del lenguaje Java**, ya sea de forma normal a partir de Java SE 8, o con anotaciones entre comentarios en versiones anteriores del lenguaje Java. Así pues, el código escrito para el Checker Framework compilará con cualquier compilador Java y se ejecutará en cualquier Máquina Virtual Java.

No obstante, **Checker Framework sólo es compatible con el OpenJDK de Oracle** y no es compatible con otros JDK de otros fabricantes. Así pues, funciona con el compilador `javac` de OpenJDK, pero no con el compilador `ecj` de Eclipse. Esto se debe a un problema que ya se comentó en los apartados de procesamiento de anotaciones: Checker Framework usa la Tree API de Oracle para navegar por los árboles sintácticos del código fuente. Los autores del framework son conscientes del problema, y preferirían una implementación más neutral, como por ejemplo a través del uso de la JSR 198 (una especificación que definía una API abierta para la navegación por árboles sintácticos y que da acceso al código de los métodos), pero desgraciadamente no hay ninguna implementación pública de esta especificación, que ha sido largamente olvidada. Así pues, las herramientas que navegan por los árboles sintácticos de código fuente Java se ven obligadas a usar la Tree API, salvo que desarrollen una API propia.

Además, como **el código de comprobación de Checker Framework es de uso opcional**, puesto que al ser sólo anotaciones, no tienen por qué utilizarse aunque estén presentes. Es decir, que se puede elegir si usar los “checkers” o ignorarlos, ya sea en todo o en parte, comprobando sólo parte del programa o ejecutando sólo determinados “checkers”.

Por otro lado, **ya existen herramientas de inferencia de tipos que ayudan o facilitan el proceso de anotar el código con anotaciones de “checkers”**. Esto se hace a través del estudio del código fuente mediante diversos algoritmos de inferencia y que ofrecen como resultado un código anotado, ya sea de forma automática o interactiva en forma de asistente. Por ejemplo: imaginemos una variable de tipo String llamada `cadena` a la que se le hace una llamada `cadena.length()` en algún punto del código analizado. Esta llamada no sería compatible con una variable `cadena` que fuese `null`, con lo que una herramienta de inferencia de código, anotaría el tipo de dicha variable como `@NotNull cadena`. Esta es una de las inferencias más sencillas, pero hay otras más complejas y que necesitan examinar más fragmentos del código para lograr asegurar otras condiciones deseables, como que sean seguras a nivel concurrente, etcétera.

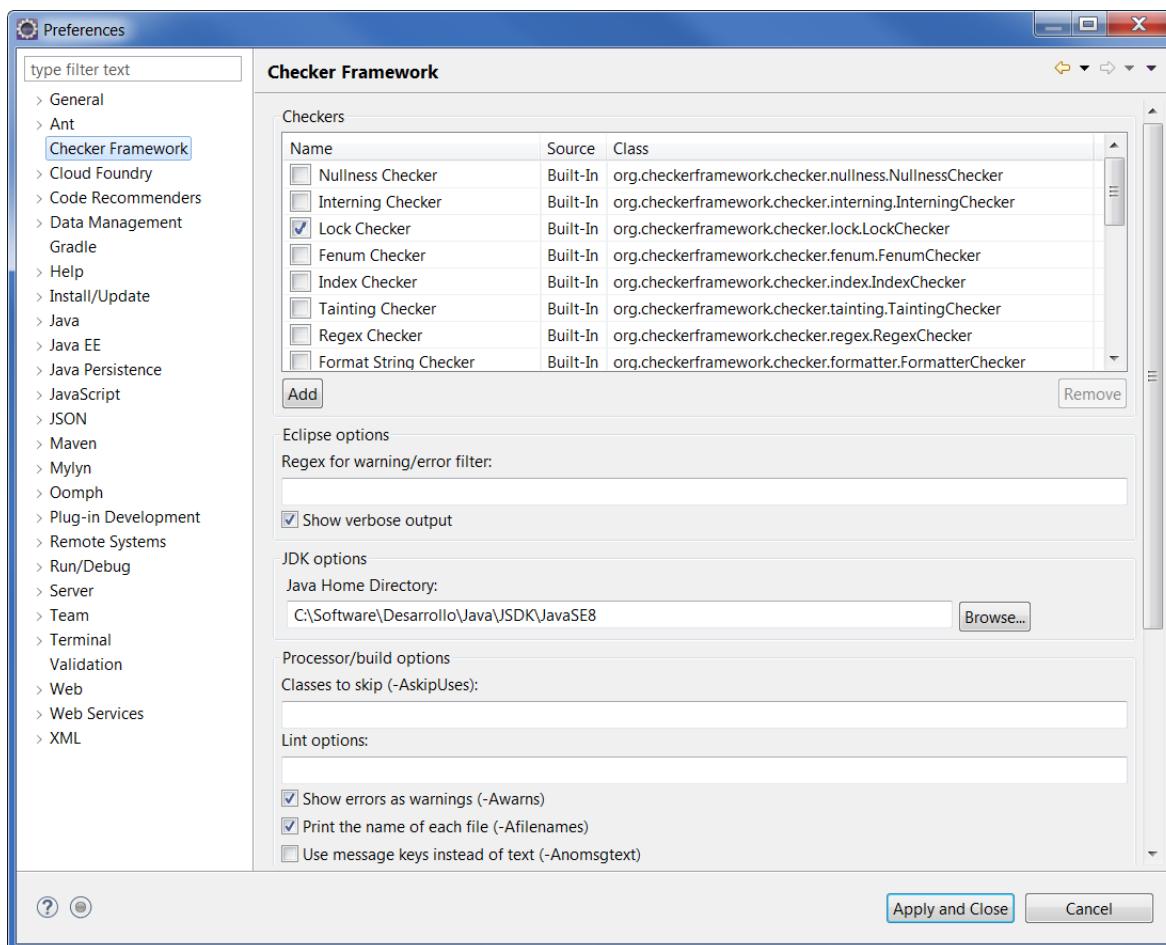
Checker Framework dispone de su propio sistema de inferencia de tipos que permite “auto-anotar” el código con checkers. Este sistema funciona mediante análisis que tratan de restringir los tipos para ser lo más restrictivos posible, pero manteniéndose compatibles con todos los usos de dichos tipos dentro del programa. Asimismo, **existen otras herramientas de inferencia compatibles con Checker Framework**, como **Cascade** [<https://github.com/reprogrammer/cascade/>], una herramienta universal de inferencia de tipos para Java SE 8.

21.3.- Plug-in para Eclipse.

Checker Framework permite la integración con Eclipse, uno de los entornos de desarrollo Java más populares, a través de un plugin específico. En este apartado, vamos a mostrar brevemente las posibilidades que ofrece este plugin.

El lector debe extrapolar estos beneficios a otras herramientas similares, ya que lo importante es tomar conciencia del gran paso adelante que supone para el desarrollo de software Java el hecho de poder disponer de estas herramientas para mejorar el diseño y la calidad del código fuente de las aplicaciones Java. Y es que este es el propósito último de la existencia de las anotaciones y de todas las tecnologías relacionadas con ellas. En resumen, lo más importante es poder llevar los beneficios de la tecnología de procesamiento de anotaciones al extremo final más cercano al desarrollador y poner estas funcionalidades a su servicio para mejorar el proceso diario de desarrollo.

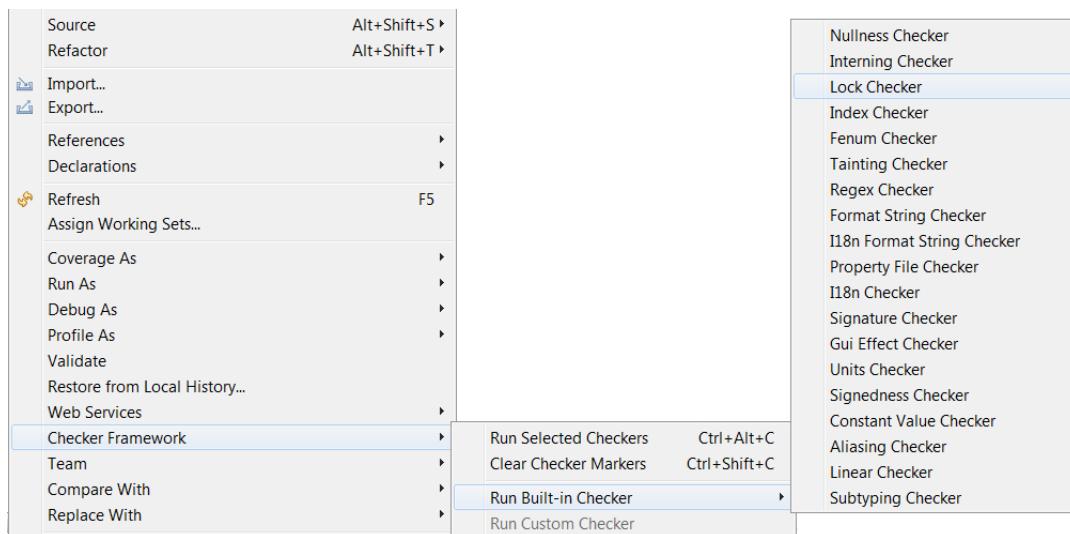
El plugin de Checker Framework se instala dentro del propio Eclipse, siguiendo las instrucciones sitas en la web oficial del framework. Una vez instalado el plugin, podremos acceder a un nuevo apartado llamado “Checker Framework” en las preferencias de Eclipse donde podremos configurar qué “checkers” tenemos seleccionados para su ejecución en bloque, así como otras opciones del plugin.



En la imagen de arriba podemos ver cómo en nuestro caso hemos seleccionado el “Lock Checker” como único checker seleccionado para este ejemplo. También hemos seleccionado la opción “Show verbose output” para tener más información sobre el funcionamiento del proceso, aunque no es necesario. Asimismo, hemos configurado manualmente el directorio home de Java, puesto que en el equipo había instaladas varios JDKs. No obstante, tampoco es necesario y, si se deja vacío, el Checker Framework usará por defecto el directorio de la JDK que se haya usado para arrancar Eclipse.

Para utilizar las clases de los “checkers” por defecto y poder ejecutar sus análisis será necesario añadir al Java Build Path del proyecto la librería “**checker-qual.jar**”. Se recomienda colocarla **en una ruta local dentro del propio proyecto**, ya que en otros directorios externos podría provocar que no se lance el análisis del plugin. Por ejemplo, puede situarse en un directorio `lib` creado a tal efecto o en el directorio `WEB-INF/lib` de las aplicaciones web.

Una vez instalado el plugin, podremos acceder a su menú contextual en la vista Project Explorer haciendo clic derecho en el ratón sobre el proyecto, paquete o clase a analizar. Se mostrará una nueva entrada del menú contextual “Checker Framework” que tendrá un contenido similar al siguiente:



Como se aprecia en la imagen, el plugin permite las siguientes opciones:

- Run Selected Checkers: ejecutar los “checkers” seleccionados en las preferencias del plugin.
- Clear Checker Markers: elimina las marcas de los warnings generados por los análisis anteriores.
- Run Built-in Checker: permite ejecutar uno de los “checkers” predefinidos en particular.
- Run Custom Checker: ejecutar un “checker” personalizado; deshabilitado si no existe ninguno.

En nuestro caso, ejecutamos sobre el código de una clase ejemplo el “Lock Checker” para comprobar si los accesos a determinadas variables guardadas por un lock son seguras a nivel concurrente.

Tras ejecutar el análisis del “Lock Checker”, el plugin del Checker Framework nos marca las líneas que contienen código detectado como potencialmente problemático con un símbolo de warning. El plugin también nos genera entradas en la vista “Problems” del entorno, una entrada por cada warning detectado en el análisis. Así pues, una vez realizado el análisis, se obtiene algo similar a esto:

The screenshot shows the Eclipse IDE interface. At the top, there is a code editor window titled "PruebaCheckerFramework.java". The code contains several annotations from the Checker Framework, such as @GuardedBy("lock") and @GuardedBy({}). Some lines are marked with yellow warning icons. Below the code editor is the "Problems" view, which displays a table of warnings. The table has columns for Description, Resource, Path, Location, and Type. There are five entries under the "Warnings (5 items)" section, all of which are "Checker Framework Problem" and occurred on line 14 or line 26.

Description	Resource	Path	Location	Type
⚠️ incompatible types in assignment.	PruebaCheckerFramework.java	/checker2/src/anotaciones/procesamiento/checker	line 14	Checker Framework Problem
⚠️ required lock not held.	PruebaCheckerFramework.java	/checker2/src/anotaciones/procesamiento/checker	line 24	Checker Framework Problem
⚠️ required lock not held.	PruebaCheckerFramework.java	/checker2/src/anotaciones/procesamiento/checker	line 25	Checker Framework Problem
⚠️ required lock not held.	PruebaCheckerFramework.java	/checker2/src/anotaciones/procesamiento/checker	line 26	Checker Framework Problem
⚠️ required lock not held.	PruebaCheckerFramework.java	/checker2/src/anotaciones/procesamiento/checker	line 26	Checker Framework Problem

Como se aprecia en la imagen, los accesos a las variables guardadas por el lock realizados antes de adquirir el lock se marcan con warnings, ya que no son seguros. Una vez adquirido el lock (línea 28), los accesos a las variables guardadas ya no se detectan como inseguros y no arrojan ningún warning por parte del “Lock Checker”.

Viendo el funcionamiento del plugin de Checker Framework para Eclipse con lo explicado en este ejemplo, el lector puede intuir el enorme potencial que herramientas como esta pueden ofrecer de cara a mejorar el día a día del desarrollo de software en cuanto a la producción de un código más robusto, seguro y de calidad.

22.- NOVEDADES EN JAVA SE 9.

Las novedades en el lenguaje Java para Java SE 9 se centran sobre todo en la modularidad y en la optimización a través del concepto de “módulo”. Los principales cambios introducidos en las APIs relacionadas con el procesamiento de anotaciones se deben a la introducción de este nuevo concepto.

22.1.- Novedades generales en Java SE 9.

El 21 de Septiembre de 2017, se liberaba **Java SE 9** desarrollada bajo la “especificación paraguas” [JSR 379](#), según el [proceso establecido en OpenJDK](#).

Las **novedades más importantes introducidas en Java SE 9** son las siguientes:

- Lenguaje Java ([JEP 213](#)):
 - try-con-recursos más conciso sin necesidad de declarar los tipos de las variables.
 - Métodos privados en interfaces (para facilitar la implementación de métodos por defecto).
 - La anotación `@SafeVarargs` se permite en métodos de instancia privados.
 - Uso de sintaxis en diamante en conjunción con las clases anidadas anónimas.
 - Se prohíbe el uso del carácter guión bajo (“_”) como identificador.
- Sistema de Módulos ([JSR 376](#)) que permite segmentar la MV Java y optimizar el uso de memoria.
- Ficheros JAR Multi-Version ([JEP 238](#)): JAR ampliados con variantes según versión de Java.
- **javac**: Compilación de código compatible con versiones anteriores de Java (desde la 6) ([JEP 247](#)).
- **jlink**: Herramienta de enlace ([JEP 282](#)) que genera imágenes ejecutables con módulos selectos.
- **applets**: Deprecados los applets ([JEP 289](#)) así como el [plug-in Java](#) para navegadores web.
- **javadoc**: Nueva Doclet API simplificada ([JEP 221](#)) y soporte para HTML5 ([JEP 224](#)).
- **javadoc**: Nueva funcionalidad de búsqueda en javadoc ([JEP 225](#)) y documentación sobre módulos.
- **JVM**: Recolección de basura: Eliminar combinaciones marcadas deprecadas en Java 8 ([JEP 214](#)).
- **JVM**: Recolección de basura: G1 como algoritmo de recolección de basura por defecto ([JEP 248](#)).
- Librerías: Anotaciones: Pipeline de anotaciones 2.0 en el compilador ([JEP 217](#)).
- Librerías: Anotaciones: Reemplazado el tag `@beaninfo` de Javadoc con una anotación adecuada, y procesar las ocurrencias de esta anotación para generar clases **BeanInfo** dinámicamente ([JEP 256](#)).
- Librerías: Anotaciones: Anotación `@Deprecated` mejorada con más elementos ([JEP 277](#)).
- Librerías: Actualización de la API de manejo de procesos del sistema operativo ([JEP 102](#)).
- Librerías: Implementación: Strings compactas para optimizar espacio en memoria ([JEP 254](#)).
- Librerías: Concurrencia: Mecanismo *publish-subscribe* y mejoras en `CompletableFuture` ([JEP 266](#)).
- Librerías: XML: Soporte de una API para catálogos OASIS XML versión 1.1 ([JEP 268](#)).
- Librerías: XML: Actualización de la implementación de la JDK a Xerces versión 2.11.0 ([JEP 255](#)).
- Librerías: Colecciones: Métodos factoría de conveniencia para colecciones ([JEP 269](#)).
- Librerías: Depuración: API de recorrido por las trazas de la pila de ejecución ([JEP 259](#)).
- Librerías: Sistema operativo: Nuevas funcionalidades de escritorio específicas ([JEP 272](#)).
- Librerías: Texto: Ficheros de propiedades con encoding UTF-8 ([JEP 226](#)).
- Librerías: Texto: Soporte de Unicode 8.0 ([JEP 267](#)).

La lista completa de mejoras introducidas en Java SE 9 puede encontrarse en la página web oficial acerca de Java SE 9 del portal de OpenJDK [<http://openjdk.java.net/projects/jdk9/>] y en el portal de OTN [<https://docs.oracle.com/javase/9/whatsnew/toc.htm>] [notas de versión].

22.2.- Novedades Java SE 9 en APIs de procesamiento.

NOTA: A veces la novedad son los tipos (clases y/o interfaces) en sí mismos, otras veces lo es la inclusión de nuevos campos y/o métodos. Para diferenciarlo claramente, los elementos nuevos aparecerán en color **verde** en los siguientes apartados.

22.2.1.- Novedades Java SE 9: API de reflexión.

Las novedades de la API de reflexión (clases `java.lang.Class`, `java.lang.Package` y las clases del paquete `java.lang.reflect`) han introducido principalmente el soporte para módulos, que están representados por instancias de la nueva clase `java.lang.Module`.

API de reflexión – Novedades Java SE 9	
Tipo	Campo(s) / Método(s)
<code>Class</code>	<code>public static Class<?> forName(Module module, String name)</code> <code>public Module getModule()</code> <code>public String getPackageName()</code>
<code>AnnotatedType</code>	<code>default AnnotatedType getAnnotatedOwnerType()</code>

22.2.2.- Novedades Java SE 9: Language Model API.

Al igual que en las APIs de reflexión, las principales novedades de la Language Model API en Java SE 9 se debe a la introducción de los módulos y sus correspondientes directivas.

Se añade la interfaz `ModuleElement` que hereda de `Element`, `QualifiedNameable` y `AnnotatedConstruct`. El soporte de los módulos en la API es similar al soporte para los paquetes. Los métodos de las interfaces generalmente usan los métodos por defecto añadidos en Java SE 8 para proporcionar una implementación nominal y mejor compatibilidad fuente con los implementadores.

Language Model API – Novedades Java SE 9	
Paquete <code>javax.lang.model</code>	
Tipo	Campo(s) / Método(s)
<code>SourceVersion</code>	<code>RELEASE_9</code> [campo que modela la versión de Java SE 9] <code>public static boolean isName(CharSequence name, SourceVersion v)</code> <code>public static boolean isKeyword(CharSequence s)</code>
Paquete <code>javax.lang.model.element</code>	
Tipo	Campo(s) / Método(s)
<code>ElementKind</code>	<code>MODULE</code> [campo que modela el tipo elemento de los módulos de Java SE 9]
<code>ModuleElement</code>	Todos sus métodos. Tiene las siguientes subinterfaces: <code>ModuleElement.Directive</code> <code>ModuleElement.DirectiveVisitor</code> <code>ModuleElement.ExportsDirective</code> <code>ModuleElementOpensDirective</code> <code>ModuleElementProvidesDirective</code> <code>ModuleElementRequiresDirective</code> <code>ModuleElementUsesDirective</code>
<code>ModuleElement.DirectiveKind</code>	Tipo enumerado con constantes para cada una de las directivas que puede contener un módulo: EXPORTS, OPENS, PROVIDES, REQUIRES y USES.
<code>UnknownDirectiveException</code>	Excepción que indica que se ha encontrado una directiva desconocida.

Language Model API – Novedades Java SE 9	
Paquete javax.lang.model.type	
Tipo	Campo(s) / Método(s)
TypeKind	MODULE [campo que modela el pseudo-tipo correspondiente a un módulo]
Paquete javax.lang.model.util	
Tipo	Campo(s) / Método(s)
Elements	<pre>default PackageElement getPackageElement(ModuleElement module, CharSequence name) default Set<? extends PackageElement> getAllPackageElements(CharSequence name) default TypeElement getTypeElement(ModuleElement module, CharSequence name) default Set<? extends TypeElement> getAllTypeElements(CharSequence name) default ModuleElement getModuleElement(CharSequence name) default Set<? extends ModuleElement> getAllModuleElements() default Elements.Origin getOrigin(Element e) default Elements.Origin getOrigin(AnnotatedConstruct c, AnnotationMirror a) default Elements.Origin getOrigin(ModuleElement m, ModuleElement.Directive dir) default boolean isBridge(ExecutableElement e) default ModuleElement getModuleOf(Element type)</pre>
ElementFilter	<pre>public static List<ModuleElement> modulesIn(Iterable<? extends Element> elements) public static Set<ModuleElement> modulesIn(Set<? extends Element> elements) public static List<ModuleElement.ExportsDirective> exportsIn(Iterable<? Extends ModuleElement.Directive> directives) public static List<ModuleElement.OpensDirective> opensIn(Iterable<? Extends ModuleElement.Directive> directives) public static List<ModuleElement.ProvidesDirective> providesIn(Iterable<? Extends ModuleElement.Directive> directives) public static List<ModuleElement.RequiresDirective> requiresIn(Iterable<? Extends ModuleElement.Directive> directives) public static List<ModuleElement.UsesDirective> usesIn(Iterable<? Extends ModuleElement.Directive> directives)</pre>
Elements.Origin	Tipo enumerado sobre el origen de los elementos: EXPLICIT, MANDATED y SYNTHETIC.

La interfaz de utilidad **Elements** tiene nuevos métodos para la búsqueda por nombre de tipos, paquetes y módulos. También se añade un para devolver el origen de una construcción del lenguaje, es decir, si ha sido implicitamente declarada en el código, sintetizada por el compilador, etcétera.

En la clase **PackageElement**, el resultado del método **getEnclosingElement** (para recuperar el elemento contenedor) depende de si el nivel de código fuente soporta módulos o no. Si no están presentes los módulos (hasta Java SE 8), el método devolverá **null**, ya que el paquete era el elemento de más alto nivel y no estaba contenido en ningún otro elemento. Si hay módulos (Java SE 9+), como elemento contenedor del paquete se devolverá el **ModuleElement** del módulo que contiene al paquete en cuestión.

Novedades de los Visitores de Valores de Anotaciones en Java SE 9

Los cambios de Java SE 9 no afectan a los visitores de valores de anotaciones. Simplemente se crean las nuevas clases relativas a la nueva versión de la plataforma Java.

Language Model API – Novedades Java SE 9 – Visitores de Valores de Anotaciones	
Paquete javax.lang.model.util	
Visitor	Método(s)
AbstractAnnotationValueVisitor9	No define ningún método nuevo.
SimpleAnnotationValueVisitor9	No define ningún método nuevo.

Novedades relacionadas con los Visitor de Elementos en Java SE 9

Los Visitor de Elementos en Java SE 9 introducen la posibilidad de visitar los nuevos elementos de los módulos en la interfaz `ElementVisitor` con el nuevo método `visitModule`.

Language Model API – Novedades Java SE 9 – Visitors de Elementos	
Paquete <code>javax.lang.model.util</code>	
Visitor	Método(s)
<code>ElementVisitor<R, P></code>	<code>default R visitModule(ModuleElement e, P p)</code>
<code>AbstractElementVisitor6<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>AbstractElementVisitor7<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>AbstractElementVisitor8<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>AbstractElementVisitor9<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>defaultAction</code>)
<code>ElementKindVisitor6<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>ElementKindVisitor7<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>ElementKindVisitor8<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>ElementKindVisitor9<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>defaultAction</code>)
<code>ElementScanner6<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>ElementScanner7<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>ElementScanner8<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>ElementScanner9<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>defaultAction</code>)
<code>SimpleElementVisitor6<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>SimpleElementVisitor7<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>SimpleElementVisitor8<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>visitUnknown</code>)
<code>SimpleElementVisitor9<R, P></code>	<code>public R visitModule(ModuleElement e, P p)</code> (que por defecto llamará a <code>defaultAction</code>)

Novedades relacionadas con los Visitor de Tipos en Java SE 9

Language Model API – Novedades Java SE 9 – Visitors de Tipos	
Paquete <code>javax.lang.model.util</code>	
Visitor	Método(s)
<code>AbstractTypeVisitor9<R, P></code>	No define ningún método nuevo.
<code>SimpleTypeVisitor9<R, P></code>	No define ningún método nuevo.
<code>TypeKindVisitor9<R, P></code>	No define ningún método nuevo.

22.2.3.- Novedades Java SE 9: Compiler Tree API.

Compiler Tree API – Novedades Java SE 9 – Árboles sintácticos de elementos de código	
Paquete com.sun.source.tree	
Tipos	Campo(s) / Método(s)
CompilationUnitTree	PackageTree getPackage()
PackageTree	Todos los métodos.
ModuleTree	Todos los métodos.
ModuleTree.ModuleKind	Tipo enumerado que define los tipos de módulos: OPEN o STRONG.
DirectiveTree	Un super-tipo para todas las directivas de un ModuleTree: ExportsTree, OpensTree, ProvidesTree, RequiresTree y UsesTree.
Paquete com.sun.source.util	
Tipos	Campo(s) / Método(s)
DocTreeFactory	Todos los métodos.
DocTrees	<pre>public abstract BreakIterator getBreakIterator() public abstract DocCommentTree getDocCommentTree(Element e) public abstract DocCommentTree getDocCommentTree(FileObject file) public abstract DocCommentTree getDocCommentTree(Element e, String relativePath) throws IOException public abstract DocTreePath getDocTreePath(FileObject fileObject, PackageElement pkgElem) public abstract List<DocTree> getFirstSentence(List<? extends DocTree> list) public abstract void setBreakIterator(BreakIterator breakiterator) public abstract DocTreeFactory getDocTreeFactory()</pre>

Compiler Tree API – Novedades Java SE 9 – Árboles sintácticos de comentarios de documentación	
Paquete com.sun.source.doctree	
Tipos	Campo(s) / Método(s)
DocCommentTree	default List<? extends DocTree> getFullBody()
HiddenTree	Todos los métodos.
IndexTree	Todos los métodos.
ProvidesTree	Todos los métodos.
UsesTree	Todos los métodos.

Toda la información de referencia completa de la Compiler Tree API para Java SE 9 puede encontrarse en la URL de la documentación oficial de la API sita en la siguiente dirección:

<https://docs.oracle.com/javase/9/docs/api/index.html?com/sun/source/tree/package-summary.html>

22.2.4.- Novedades Java SE 9: API de procesamiento JSR 269.

Las novedades en la API de procesamiento JSR 269 para Java SE 9 se pueden ver en su revisión de mantenimiento 3, disponible en el [registro de cambios de la especificación](#), donde puede consultarse un [resumen de los principales cambios](#), que presentamos a continuación.

Como en el caso de las demás APIs que modelan o procesan elementos del código fuente del lenguaje Java, la API de procesamiento JSR 269 ha tenido que adaptarse al nuevo concepto de módulos. Además de eso, también se han añadido algunos métodos nuevos por conveniencia y otros ajustes.

Los ficheros de entrada iniciales para el procesamiento de anotaciones ahora pueden incluir ficheros “module-info.java”. Estos ficheros se representan con objetos de tipo `ModuleElement`. Sin embargo, no se pueden crear ficheros de código o de clase de un nuevo elemento módulo a través de la API de la interfaz `Filer`. En ella, la sintaxis de los parámetros “`name`” de los métodos de creación ha sido expandida para permitir que empiecen con el prefijo “`nombre-modulo/`”.

La sintaxis del método `Processor.getSupportedAnnotationTypes` también ha sido expandida para permitir que empiece por un nombre de módulo y una barra.

Para los procesadores que tengan que ejecutarse tanto antes como después de Java SE 9, es decir, tanto sin módulos como con ellos, la recomendación es que los nombres de los tipos de anotación soportados incluyan el prefijo del módulo, "**nombre-modulo/**", y descansen en el código del método `AbstractProcessor.getSupportedAnnotationTypes` para quitarle dicho nombre del módulo cuando se esté usando un nivel de código fuente anterior a Java SE 9.

Para permitir que se puedan buscar múltiples tipos anotación en una sola llamada, se han añadido nuevos métodos sobrecargados `RoundEnvironment.getElementsAnnotatedWithAny`. De esta forma, no hay que hacer múltiples llamadas a `RoundEnvironment.getElementsAnnotatedWith` y acumular sus resultados.

Finalmente, comentar que la anotación `@Generated`, que indica en un código fuente que este ha sido generado mediante un proceso de generación de código fuente, ha sido deprecada de su paquete `javax.annotation` y se ha creado otra homónima en `javax.annotation.processing`, ya que se ha considerado que este es el paquete más adecuado para ubicarla.

API de procesamiento JSR 269 – Novedades Java SE 9	
Paquete javax.annotation.processing	
Tipos	Campo(s) / Método(s)
RoundEnvironment	<pre>default Set<? extends Element> getElementsAnnotatedWithAny(TypeElement... annotations) default Set<? extends Element> getElementsAnnotatedWithAny(Set<Class<? extends Annotation>> annotations)</pre>
@Generated	Todos los métodos. Reemplazo de la anotación homónima de javax.annotation.

22.3.- Novedades del procesamiento en Java SE 9.

22.3.1.- Pipeline de anotaciones 2.0 en el compilador (JEP 217).

Como hemos visto, Java SE 8 introdujo dos nuevas funcionalidades relacionadas con las anotaciones: Anotaciones Repetibles (JEP 120) y Anotaciones sobre Usos de Tipos (JSR 308 / JEP 104). Además, las Expresiones Lambda (JSR 335 / JEP 126) añadieron nuevas posiciones sintácticas donde poder colocar anotaciones en nuevas posiciones sintácticas. Estas funcionalidades, ninguna de las cuales existían anteriormente, pueden ser combinadas, conduciendo a líneas de código como por ejemplo `Function<String, String> fss = (@Anotacion @Anotacion String s) -> s;`

El canal (“pipeline”) de procesamiento existente del compilador `javac` no podía manejar estos casos directamente y, como resultado de ello, su diseño original se estiró para acomodar los nuevos casos de uso, conduciendo a una implementación frágil y difícil de mantener.

En el apartado de “Procesamiento en Java SE 8” ya explicamos que las APIs relacionadas con el procesamiento de anotaciones no habían sido correctamente actualizadas para manejar las nuevas anotaciones sobre usos de tipos. Obviamente, esto también tiene que ver con el hecho de que la arquitectura del compilador `javac` y sus APIs y herramientas fueron diseñadas hace más de dos décadas y están muy desfasadas.

Para Java SE 9 se ha realizado un proceso de renovación del procesamiento de anotaciones por parte del compilador a través del proyecto “**Annotations Pipeline 2.0**” ([JEP 217](#)). Su objetivo es rediseñar el canal (“pipeline”) por el que se procesan las anotaciones en `javac` para que se ajuste mejor a los requisitos de las anotaciones y de las herramientas que procesan anotaciones.

La meta del proyecto “Annotations Pipeline 2.0” es reemplazar esta vieja arquitectura con una nueva que soporte los nuevos casos de uso de una forma más integrada, conduciendo a un código más correcto y fácil de mantener.

Los **principales objetivos del proyecto “Annotations Pipeline 2.0”** son los siguientes:

- Lograr que el compilador se comporte correctamente con respecto a las anotaciones, ya sean sobre declaraciones o sobre usos de tipos. Además, los ficheros de clase emitidos deberían tener sus atributos correctamente formateados para todas las clases de anotaciones.
- Las APIs de “reflexión en tiempo de compilación” (las de los paquetes `javax.lang.model` y `javax.annotation.processing`) deberían manejar adecuadamente todas las anotaciones en posiciones de signatura visibles.
- La API de reflexión en tiempo de ejecución debería funcionar correctamente extrayendo información de las anotaciones desde los ficheros de clase.

Se pretende que la refactorización del canal de procesamiento de las anotaciones no provoque cambios exteriores perceptibles, excepto en lo que se refiere a mejorar la corrección y corregir los bugs. Todo el trabajo se ha realizado a través del [proyecto Annotations Pipeline 2.0 en OpenJDK](#).

23.- CONCLUSIONES.

Cuando se empezó este proyecto, el objetivo fundamental era dar a conocer todos los aspectos de las anotaciones de código del lenguaje Java a través de una exposición de contenidos estructurada y exhaustiva. La motivación detrás de la elección de este objetivo era paliar en la medida de lo posible la falta de documentación sobre su funcionamiento más allá de su uso normal en el código cliente.

Esta falta de documentación crea una “barrera de entrada” para los desarrolladores, que son obligados a invertir mucho tiempo en investigación y en procesos de prueba y error para averiguar cómo se ha de trabajar con las APIs. Dicha información, obtenida tras invertir mucho tiempo en descubrir cada uno de los detalles del proceso, es la que se ha incluido en el manual, facilitando una mejor comprensión de su funcionamiento y reduciendo drásticamente el periodo de adopción.

Las tres APIs relacionadas con el procesamiento de anotaciones (Language Model / JSR 269 API, Core Reflection y Compiler Tree API) tienen ciertos problemas de diseño (especialmente a la hora de permitir el *testing*), pero en la práctica permiten implementar la mayoría de casos de uso.

No obstante, su implementación no cumple con ciertos requisitos y tiene algunos errores. Por ejemplo, a fecha actual con Java SE 9, ninguna de las tres APIs permite descubrir y procesar todos los tipos de anotaciones, especialmente las tan cacareadas anotaciones sobre usos de tipos introducidas en Java SE 8. Tampoco es posible combinar dichas APIs, ya que apenas son interoperables entre sí. Esto provoca que no exista ninguna API oficial que nos permita descubrir y procesar todos los tipos de anotaciones, dándose la paradoja de que podemos colocar anotaciones que el compilador detecte como legales, pero con las que no podríamos hacer nada útil, al no poder descubrirlas y procesarlas. Será necesario esperar a futuras versiones de Java en las que se corrijan todos estos problemas.

Debido a estos problemas y limitaciones de las APIs de procesamiento de anotaciones, Oracle recomienda a los desarrolladores usuarios de sus APIs que se construyan sus propias herramientas, librerías o clases de utilidad para el procesamiento. Esto hace que los ejemplos proporcionados en el ámbito de este proyecto no sean tan elegantes como deberían, pero al menos las clases auxiliares que se proporcionan sí permiten descubrir y procesar un buen número de tipos de anotaciones.

Así pues, el presente manual permite no sólo aprender y dominar el uso, análisis y diseño de tipos anotación, o integrar distintas herramientas en los entornos de desarrollo para facilitar el proceso, si no que ayuda a amortiguar en la medida de lo posible los problemas de la implementación del procesamiento de anotaciones que hemos descrito a través de la información y los códigos fuente de ejemplo que contiene, por lo que, en ausencia de información fehaciente y herramientas bien diseñadas, resultará una ayuda fundamental para alumnos, investigadores y desarrolladores.

A pesar de los problemas indicados, las anotaciones han hecho efectivo su enorme potencial, siendo una de las características del lenguaje Java más utilizadas en la actualidad y que más se han desarrollado. Y las perspectivas de futuro indican que el uso de las anotaciones, ya muy extendido en la actualidad, todavía se va a seguir potenciando aún más en el futuro.

Como ya hemos explicado cuando hablamos del Checker Framework, la principal tendencia para los usos futuros de las anotaciones es el aprovechamiento de la información que proporcionan sus meta-datos para ampliar el sistema de tipos de Java y poder maximizar la detección de errores en tiempo de código fuente, mejorando así la calidad del software producido.

24.- BIBLIOGRAFÍA.

Desgraciadamente, es escasa la información acerca del diseño de anotaciones propias y el uso de la API de la especificación [JSR 269](#) (Pluggable Annotation Processing API), que es la que hay que utilizar para implementar el procesamiento de anotaciones.

Esta carencia tan grave de información y documentación sobre las anotaciones en Java hace que estas se vean por parte de muchos desarrolladores como algo lejano y que, en general, sean poco utilizadas, poco tenidas en cuenta, poco conocidas o incluso desconocidas totalmente por parte de algunos desarrolladores novatos. Esto supone una importante pérdida en cuanto a la calidad del código generado por dichos desarrolladores que, gracias al uso de las anotaciones, podría ser más conciso, legible, de mayor calidad y podría haberse realizado en menos tiempo, incrementando la productividad.

En esta sección se recopilan una serie de referencias bibliográficas y recursos adicionales sobre las herramientas y tecnologías tratadas en este documento. Haciendo uso de ellas, el lector podrá ampliar información y profundizar en los conceptos que más le interesen.

24.1.- Libros.

Título: Piensa en Java. 4^a edición.

Autor: Bruce Eckel.

Editorial: Prentice Hall. Pearson Educación, S. A. (Madrid).

Fecha: 2007.

I.S.B.N.: 978-84-896-6034-2.

Nº páginas: 1004.

Descripción: Excelente libro de referencia, considerado por muchos como la “Biblia” *de facto* para la iniciación de desarrolladores principiantes a los fundamentos del lenguaje Java. Uno de los títulos más completos y accesibles, aunque muy caro en su edición española. La 4^a edición está actualizada a Java SE versión 6 (salvo en lo que respecta precisamente a las anotaciones).

Sobre Anotaciones: Dedica el capítulo 20 a las anotaciones, reservándoles la extensión que merecen. Explica de forma bastante completa (aunque superficial en última instancia dado que se trata de un título introductorio), el antiguo método de trabajo con las anotaciones.

Título: Profesional Java JDK 6.

Autores: Clay Richardson, Donald Avondolio, Scot Schrager, Mark Mitchell, Jeff Scanlon.

Editorial: Anaya Multimedia. Grupo Anaya, S. A. (Madrid).

Fecha: 2007.

I.S.B.N.: 978-84-415-2220-6.

Nº páginas: 800.

Descripción: Libro complementario del «Piensa en Java», con un enfoque más práctico. Escrito por programadores para programadores y va directamente al grano ofreciendo soluciones.

Sobre Anotaciones: En el capítulo 1, apartado “Nuevas característica de lenguaje de Java 5”, dedica unas pocas páginas a hablar de las anotaciones bajo el epígrafe “Metadatos”. Una mención realmente insignificante y que parece más bien enfocada a “cubrir el expediente”.

24.2.- Páginas web.

Esta sección contiene referencias a diversos recursos web: primeramente, algunos que vendrán muy bien como referencia rápida, a continuación recursos básicos de desarrollo fundamentales para poder empezar a trabajar con anotaciones, así como referencias a la documentación técnica oficial disponible más importante.

Además, se incluye una selección de artículos y tutoriales no oficiales, escogidos por su alto nivel de calidad y relevancia, que pueden resultar interesantes como lectura adicional al contenido de este manual.

24.2.1.- Páginas generales de referencia.

Plataforma de desarrollo Java @ Wikipedia

[http://en.wikipedia.org/wiki/Java_\(software_platform\)](http://en.wikipedia.org/wiki/Java_(software_platform))

Java SE (Java Platform, Standard Edition) @ Wikipedia

http://en.wikipedia.org/wiki/Java_SE

JDK (Java Development Kit) @ Wikipedia

http://en.wikipedia.org/wiki/Java_Development_Kit

Anotaciones Java @ Wikipedia

http://en.wikipedia.org/wiki/Java_annotation

http://es.wikipedia.org/wiki/Anotación_Java

24.2.2.- Recursos de desarrollo.

Java SE @ Oracle TechNetwork (OTN)

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

Entorno de Desarrollo Eclipse

<http://www.eclipse.org>

Entorno de Desarrollo NetBeans

<https://netbeans.org>

Entorno de Desarrollo IntelliJ IDEA

<https://www.jetbrains.com/idea/>

24.2.3.- Documentación técnica básica.

Java SE 8 API @ docs.oracle.com

<https://docs.oracle.com/javase/7/docs/api/>

<http://docs.oracle.com/javase/8/docs/api/>

<https://docs.oracle.com/javase/9/docs/api/index.html?overview-summary.html>

JSR 269: API javax.annotation.processing @ docs.oracle.com

<http://docs.oracle.com/javase/7/docs/api/index.html?javax/annotation/processing/package-summary.html>

<https://docs.oracle.com/javase/8/docs/api/index.html?javax/annotation/processing/package-summary.html>

<https://docs.oracle.com/javase/9/docs/api/index.html?javax/annotation/processing/package-summary.html>

Compiler Tree API @ docs.oracle.com

<https://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/>

<https://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/>

<https://docs.oracle.com/javase/9/docs/api/index.html?com/sun/source/tree/package-summary.html>

Java SE Technotes: Annotations @ docs.oracle.com

<https://docs.oracle.com/javase/7/docs/technotes/guides/language/annotations.html>

<http://docs.oracle.com/javase/8/docs/technotes/guides/language/annotations.html>

Java Tutorials: Annotations

<http://docs.oracle.com/javase/tutorial/java/annotations/>

The Java Language Specification – Java SE 8 Edition @ docs.oracle.com

<http://docs.oracle.com/javase/specs/jls/se8/html/index.html>

Apartado 9.6: Annotation Types:

<http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.6>

Apartado 9.7: Annotations:

<http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.7>

JSR 269: Pluggable Annotation Processing API @ JCP.org

<http://www.jcp.org/en/jsr/detail?id=269>

JSR 308: Making Java annotations more general and more useful – Presentación resumen

<https://jcp.org/aboutJava/communityprocess/ec-public/materials/2012-01-1011/jsr308-201201.pdf>

JSR 308: Annotations on Java Types

<https://jcp.org/en/jsr/detail?id=308>

Annotations on Java Types – JSR 308 Expert Group

<http://cr.openjdk.java.net/~abuckley/308.pdf>

Type Annotations (JSR 308) and the Checker Framework

<http://types.cs.washington.edu/jsr308/>

The Java Programming Language Compiler @ docs.oracle.com

<http://docs.oracle.com/javase/8/docs/technotes/guides/javac/>

24.2.4.- Documentación técnica adicional.

Joseph Darcy's blog – Entradas sobre procesamiento de anotaciones por su ingeniero jefe
<https://blogs.oracle.com/darcy/annotation-processing>

Java Infrastructure Developer's guide (AKA Concepts & HowTos) @ NetBeans.org
http://wiki.netbeans.org/Java_DevelopersGuide

JSR 308 Explained: Java Type Annotations

<https://jaxenter.com/jsr-308-explained-java-type-annotations-107706.html>

<http://www.oracle.com/technetwork/articles/java/ma14-architect-annotations-2177655.html>

Lista de correo “type-annotations-spec-experts”

<http://mail.openjdk.java.net/pipermail/type-annotations-spec-experts/>

Lista de correo “type-annotations-spec-comments”

<http://mail.openjdk.java.net/pipermail/type-annotations-spec-comments/>

Lista de correo “type-annotations-spec-observers”

<http://mail.openjdk.java.net/pipermail/type-annotations-spec-observers/>

Lista de correo “type-annotations-dev”

<http://mail.openjdk.java.net/pipermail/type-annotations-dev/>

Lista de correo “compiler-dev”

<http://mail.openjdk.java.net/pipermail/compiler-dev/>

The Hitchhiker's Guide to javac @ OpenJDK

<http://openjdk.java.net/groups/compiler/doc/hhgtjavac/>

Java SE Compiler API @ JavaBeat

<http://www.javabeat.net/2007/04/the-java-6-o-compiler-api/>

Java Annotation Processing Tool (apt) (a extinguir / deprecada)

<http://docs.oracle.com/javase/7/docs/technotes/guides/apt/>

Java Mirror API @ docs.oracle.com (a extinguir / deprecada)

(Modela el lenguaje Java y está pensado para su uso con procesadores de anotaciones J2SE 1.5)

<http://docs.oracle.com/javase/7/docs/jdk/api/apt/mirror/index.html?overview-summary.html>

24.2.5.- Artículos.

Annotations in Java 5.0 @ JavaBeat

<http://www.javabeat.net/annotations-in-java-5-0/>

Java Annotations: An Introduction @ developer.com

<http://www.developer.com/java/other/article.php/3556176/An-Introduction-to-Java-Annotations.htm>

Introduction to Java 6.0 New Features, Part-1 @ JavaBeat

<http://www.javabeat.net/2007/06/introduction-to-java-6-0-new-features-part-i/>

Java 6.0 Features Part – 2: Pluggable Annotation Processing API @ JavaBeat

<http://www.javabeat.net/2007/06/java-6-0-features-part-2-pluggable-annotation-processing-api/>

Principles for using annotations

<http://cyrille.martraire.com/2009/12/principles-for-using-annotations/>

The Advanced Compiler Java API – Part 1 @ Java Magazine

http://www.oraclejavamagazine-digital.com/javamagazine_open/20130304#pg44

Java Service Provider Interface (SPI): Creating Extensible Java Applications @ developer.com

<http://www.developer.com/java/article.php/3848881/Service-Provider-Interface-Creating-Extensible-Java-Applications.htm>

Patterns for using custom annotations

<http://cyrille.martraire.com/2010/07/patterns-for-using-annotations/>

Java 8's new Type Annotations @ blogs.oracle.com

<https://blogs.oracle.com/java-platform-group/java-8s-new-type-annotations>

Annotate your code to find more bugs

<http://jetbrains.dzone.com/articles/find-bugs-your-code>

Type Annotations in Java 8: Tools and Opportunities

<https://www.infoq.com/articles/Type-Annotations-in-Java-8>

Java Annotations and Reflection: Powerful Building Blocks for a DBMS Interface

<http://java.dzone.com/articles/java-annotations-and>

24.2.6.- Tutoriales.

Annotation Processing 101 by Hannes Dorfmann

<http://hannesdorfmann.com/annotation-processing/annotationprocessing101>

Code Generation using Annotation Processors in the Java language @ dr. macphail's trance

Part 1: <http://deors.wordpress.com/2011/09/26/annotation-types/>

Part 2: <http://deors.wordpress.com/2011/10/08/annotation-processors/>

Part 3: <http://deors.wordpress.com/2011/10/31/annotation-generators/>

25.- HISTORIAL DE CAMBIOS.

Como referencia respecto a la evolución y mantenimiento del presente manual, aquí se deja constancia de los principales cambios producidos a lo largo de sus sucesivas versiones.

Versión 1.1 (5 de Diciembre de 2017)

- Añadido un apartado “Conclusiones”.

Versión 1.0 (22 de Noviembre de 2017)

- Primera versión completa del manual.
- Revisión final de todos los contenidos para una mejor comprensión.
- Completado el apartado “Novedades en Java SE 9”.
- Completado el apartado “Procesamiento en Java SE 8”.
- Completado el apartado “Checker Framework”.
- Completado el apartado “Transformación de código fuente”.

Versión 0.95 (31 de Agosto de 2015)

- Nuevo apartado “Acerca del manual”.
- Nuevo apartado “Conceptos básicos”.
- Nuevo apartado “Análisis y diseño de tipos anotación”.
- Completado el apartado “Anotaciones sobre tipos (JSR 308)”.
- Restructuración de los apartados para seguir una mejor evolución cronológica.

Versión 0.9 (11 de Junio de 2014)

- Completado el macro-apartado “Procesamiento JSR 269 (Java SE 6+)”.
- Añadidos apartados dedicados a la Java Compiler API (JSR 199) y la Compiler Tree API. Compiler Tree API nos permitirá descubrir anotaciones sobre variables locales (!!).

Versión 0.8 (26 de Mayo de 2014)

- Completado el macro-apartado “Procesamiento J2SE 1.5”.
- Revisión general de toda la redacción para una mejor comprensión.
- Añadido apartado “Principios de diseño para el uso de anotaciones”.
- Reescrito el apartado “Anotaciones en Java SE 8 (JSR 308)”.
- Añadida @Repeatable al apartado “Meta-Anotaciones predefinidas”.
- Añadida @FunctionalInterface al apartado “Anotaciones predefinidas”.

Versión 0.7 (25 de Abril de 2014)

- Añadido apartado “Commons Annotations (JSR 250)”.
- Añadidos cuadros resumen de referencia para todos los tipos anotación explicados.
- Separados los apartados “Anotaciones predefinidas” y “Meta-Anotaciones predefinidas”.
- Añadida @SafeVarargs al apartado “Anotaciones predefinidas”.
- Añadidas @Inherited y @Documented al apartado “Meta-Anotaciones predefinidas”.

Versión 0.6 (5 de Noviembre de 2013)

- Añadido apartado “Anotaciones repetibles (Java SE 8)”.
- Añadido apartado “Herramientas de manipulación de ficheros de clase”, con un ejemplo que muestra como recuperar anotaciones con retención CLASS usando Javassist.

Versión 0.5 (15 de Septiembre de 2013)

- Añadido apartado general Procesamiento de anotaciones, hablando de los diferentes métodos de procesamiento de las anotaciones según su política de retención.
- Añadidas secciones sobre el procesamiento de anotaciones con Java SE 8, hablando de la nueva especificación JSR 308 (Annotations on Java Types) y el Checker Framework.

Versión 0.4 (9 de Septiembre de 2013)

- Se separa el amplio contenido histórico de la Introducción en Historia de las anotaciones.
- Se separa todo el contenido referente al procesamiento de anotaciones, mucho más amplio de lo esperado, en varias secciones para tratarlo de forma más pormenorizada.
- Separadas secciones Anotaciones, Tipos de anotación y Tipos de anotación predefinidos debido a la ampliación y restructuración de sus contenidos.

Versión 0.3 (Sábado 31 de Agosto de 2013)

- Re-estructuración de secciones y ampliación de contenidos siguiendo las directrices del tutor académico del proyecto: se concentran los contenidos teóricos en la introducción, y se amplían las secciones más prácticas, acompañadas de nuevos ejemplos aclaratorios.
- Revisión de la redacción y los ejemplos de código para una mejor comprensión del lector.
- Finalizado el proceso de revisión inicial de documentación.

Versión 0.2 (Domingo 18 de Agosto de 2013)

- Completadas las secciones de introducción, tipos anotación y anotaciones.
- Se revisa el esqueleto inicial con secciones y, sobre todo, muchos apartados nuevos.

Versión 0.1 (Domingo 28 de Julio de 2013)

- Versión inicial del documento. Esqueleto de contenidos. Revisión de documentación.