Técnicas Avanzadas de Diseño de Software:

Una introducción a la Programación Orientada a Objetos usando *UML* y *Java*

Autor:

José F. Vélez Serrano

Colaboradores:

Angel Sánchez

Almudena Sierra

Isidoro Hernán

Alberto Peña

Santiago Doblas

Alfredo Casado

Francisco Gortázar

Prólogo

El presente texto surge de varios años de experiencia docente de los autores en las asignaturas "*Software* Avanzado" y "Lenguajes Informáticos", que se imparten respectivamente en el tercer curso de la carrera de Ingeniería Técnica de Informática de Gestión y en el segundo curso de la carrera Ingeniería Informática, en la Escuela Técnica Superior de Ingeniería Informática de la Universidad Rey Juan Carlos de Madrid.

El objetivo de este texto es la introducción del paradigma de la programación orientada a objetos, del diseño basado en patrones, del Lenguaje Unificado de Modelado (*UML*) y del lenguaje de programación *Java* en su versión 5. Con estas bases se pretende que el lector consiga unos conocimientos teóricos en estas materias y a la vez pueda comprobar su utilidad práctica.

El texto se puede leer de manera continua, pero también se puede leer como una introducción a la Programación Orientada a Objetos (si sólo se leen los capítulos 1, 5 y 8), o como una introducción al lenguaje *Java* (si sólo se leen los capítulos 2, 3, 4, 6 y 7). Cabe decir que al terminar cada capítulo se plantean algunos ejercicios que se encuentran resueltos en el anexo A.

Respecto al formato del texto se han tomado varias decisiones. Para empezar, cuando se introduce un nuevo concepto se escribe en **negrita** y se añade al glosario que se encuentra en el anexo B. Además, debido al carácter de la materia, el texto está salpicado de anglicismos y marcas (que se han escrito en *cursiva*), de referencias a código en el texto (que se han escrito en fuente `courier`) y de fragmentos de programas (que, también en *courier*, se han `sombreado`).

Índice

Capítulo 1 Introducción a la programación orientada a objetos 1

1.1. Complejidad del software: origen y tratamiento 1
1.1.1 Herramientas para tratar la complejidad 1
1.1.2 Evolución de los lenguajes de programación 2
1.1.3 Paradigmas de programación 3
1.2. Programación orientada a objetos 3
1.2.1 Abstraer 4
1.2.2 Encapsular 5
1.2.3 Jerarquizar 6
1.2.4 Modularizar 10
1.2.5 Tipo 11
1.2.6 Concurrencia 13
1.2.7 Persistencia 13
1.3. Lecturas recomendadas 14
1.4. Ejercicios 14

Capítulo 2 Introducción al lenguaje Java 15

2.1. Introducción 15
2.1.1 Breve historia de Java 15
2.1.2 La máquina virtual de Java 16
2.1.3 Compiladores de Java 16
2.1.4 Entornos de desarrollo para Java 16
2.1.5 El programa hola mundo 16
2.1.6 Los comentarios 17
2.1.7 Juego de instrucciones 18
2.2. Expresiones básicas y control de flujo 18
2.2.1 Tipos primitivos 18
2.2.2 Conversiones entre tipos primitivos 19
2.2.3 Operadores 20
2.2.4 Control del flujo 21
2.3. Las clases 23
2.3.1 Las propiedades en detalle 25
2.3.2 Los métodos en detalle 27
2.3.3 Creación de objetos 28
2.3.4 Destrucción de objetos 29
2.3.5 Bloques de inicialización 29
2.3.6 Los modificadores de control de acceso a los miembros en detalle 30
2.3.7 Los modificadores de uso de los miembros en detalle 30
2.3.8 Modificadores en la definición de clases generales 31
2.3.9 Clases internas 31
2.3.10 Ejemplo del parking 32
2.3.11 Definición de arrays de objetos y de tipos primitivos 33
2.4. Los paquetes 34
2.5. Lecturas recomendadas 35
2.6. Ejercicios 35

Capítulo 3 Herencia y genericidad en Java 37

3.1. La herencia de clases ... 37
3.1.1 Herencia en clases internas ... 38
3.1.2 Redefinición de miembros ... 38
3.1.3 La herencia y los constructores ... 38
3.2. Polimorfismo dinámico ... 39
3.2.1 El casting de referencias a objetos ... 41
3.3. Las interfaces ... 42
3.4. La genericidad ... 44
3.4.1 Clases e interfaces genéricas ... 44
3.4.2 Métodos con tipos parametrizados ... 47
3.5. Conclusiones ... 47
3.6. Lecturas recomendadas ... 47
3.7. Ejercicios ... 47

Capítulo 4 El paquete java.lang ... 49

4.1. La clase Object ... 49
4.1.1 La clase ClassLoader ... 49
4.2. Clases para el manejo de cadenas de caracteres ... 50
4.3. Las excepciones ... 51
4.3.1 Tipos de excepciones ... 52
4.3.2 El uso práctico de las excepciones ... 53
4.4. Sobrecarga de los métodos básicos de object ... 55
4.4.1 Interfaz Cloneable ... 55
4.4.2 Los métodos equals y hashCode ... 56
4.4.3 Interfaz Comparable ... 57
4.5. La concurrencia ... 57
4.5.1 La palabra reservada synchronized ... 58
4.5.2 Comunicación entre hilos ... 58
4.6. Los tipos enumerados ... 59
4.7. Envolturas ... 61
4.8. Lecturas recomendadas ... 62
4.9. Ejercicios ... 62

Capítulo 5 Diseño de Clases ... 63

5.1. Diseño top-down apoyado en UML ... 63
5.1.1 Relaciones de dependencia entre objetos ... 64
5.1.2 Relaciones de asociación entre objetos ... 67
5.1.3 Relaciones entre clases ... 71
5.2. Conclusiones ... 76
5.3. Lecturas recomendadas ... 76
5.4. Ejercicios ... 76

Capítulo 6 Entrada salida en Java ... 79

6.1. Streams de bytes ... 79
6.2. Streams de caracteres ... 81
6.2.1 Streams del sistema ... 83
6.2.2 StreamTokenizer y Scanner ... 83
6.3. Manejo de ficheros ... 84
6.4. La interfaz Serializable ... 84
6.5. Conclusiones ... 85

6.6. Ejercicios ... 85

Capítulo 7 Estructuras de datos predefinidas en Java ... 87

7.1. Collection ... 87
7.1.1 El EnumSet ... 88
7.2. Map ... 89
7.3. Iteradores ... 90
7.4. La clase Collections ... 90
7.5. El resto del paquete java.util ... 91
7.6. Conclusiones ... 91
7.7. Lecturas recomendadas ... 91
7.8. Ejercicios ... 91

Capítulo 8 Patrones y principios de diseño ... 93

8.1. Principales Patrones de Diseño ... 93
8.1.1 Fábrica Abstracta (Abstract Factory) ... 93
8.1.2 Adaptador o Envoltorio (Adapter o Wrapper) ... 95
8.1.3 Decorador (Decorator) ... 95
8.1.4 Composición (Composite) ... 97
8.1.5 Iterador (Iterator) ... 97
8.1.6 Estrategia (Strategy) ... 97
8.1.7 Comando (Command) ... 98
8.1.8 Observador (Observer) ... 100
8.2. Algunos principios útiles en POO ... 100
8.2.1 El principio abierto-cerrado ... 100
8.2.2 El principio de Liskov ... 101
8.2.3 El principio de segregación de interfaces ... 101
8.2.4 Maximizar el uso de clases inmutables ... 101
8.2.5 Preferir composición frente a herencia ... 101
8.2.6 Principio de responsabilidad única ... 101
8.2.7 Eliminar duplicaciones ... 101
8.2.8 Principio de inversión de dependencia ... 101
8.2.9 Preferir el polimorfismo a los bloques if/else o switch/case ... 101
8.2.10 Conclusiones ... 102
8.3. Lecturas recomendadas ... 102
8.4. Ejercicios ... 102

Anexo A Solución de los ejercicios ... 105

A.1 Ejercicios del capítulo 1 ... 105
A.1.1 Solución del ejercicio 1 ... 105
A.2 Ejercicios del capítulo 2 ... 106
A.2.1 Solución del ejercicio 1 ... 106
A.2.2 Solución del ejercicio 2 ... 106
A.2.3 Solución del ejercicio 3 ... 106
A.3 Ejercicios del capítulo 3 ... 106
A.3.1 Solución del ejercicio 1 ... 106
A.3.2 Solución del ejercicio 2 ... 107
A.3.3 Solución del ejercicio 3 ... 107
A.4 Ejercicios del capítulo 4 ... 108
A.4.1 Solución del ejercicio 1 ... 108
A.5 Ejercicios del capítulo 5 ... 109

A.5.1 Solución del ejercicio 1 109
A.5.2 Solución del ejercicio 2 109
A.5.3 Solución del ejercicio 3 112
A.5.4 Solución del ejercicio 4 114
A.5.5 Solución del ejercicio 5 118
A.6 Ejercicios del capítulo 6 121
A.6.1 Solución del ejercicio 1 121
A.7 Ejercicios del capítulo 7 123
A.7.1 Solución del ejercicio 1 123

Anexo B Índice alfabético 125

Capítulo 1 Introducción a la programación orientada a objetos

En este primer capítulo se comienza analizando las características que hacen complejo el *software* y las herramientas que habitualmente se utilizan para enfrentar esa complejidad. Después, se repasa la evolución de los lenguajes de programación a través del tiempo, y se presentan los principales paradigmas de programación. Finalmente se analiza el paradigma de orientación a objetos, describiendo los mecanismos que debe proporcionar un lenguaje de programación orientado a objetos.

1.1. Complejidad del *software*: origen y tratamiento

En el desarrollo de un proyecto de *software* de tamaño medio o grande suelen intervenir varias personas que toman múltiples decisiones, tanto en el diseño como en la implementación. Normalmente, como resultado se obtiene un conjunto de programas y bibliotecas con cientos de variables y miles de líneas de código. Cuando se producen errores en el desarrollo de un proyecto de este tipo, o cuando el proyecto crece hasta hacerse incomprensible, surgen las preguntas: ¿qué hacemos mal?, ¿por qué es tan complejo este sistema o este desarrollo?

Para responder a estas preguntas hay que empezar por aceptar que la complejidad es una propiedad inherente al *software* y no un accidente debido a una mala gestión o a un mal diseño. Según Grady Booch, la complejidad en el *software* y en su desarrollo se deriva fundamentalmente de los siguientes cuatro elementos:

- **La complejidad del dominio del problema.-** Un proyecto *software* siempre está salpicado por la complejidad propia del problema que pretende resolver. Por ejemplo, el desarrollo de un *software* de contabilidad tiene la complejidad propia del desarrollo, más la complejidad de las normas y del proceso de contabilidad.
- **La dificultad de gestionar el proceso de desarrollo.-** Cuando el desarrollo *software* que se realiza tiene miles de líneas de código, cientos de ficheros, muchos desarrolladores... el proceso que gestiona todos estos elementos no es trivial.
- **La flexibilidad de las herramientas de *software*.-** Esta flexibilidad es un obstáculo, ya que permite a los desarrolladores usar elementos muy básicos para construir el *software* desde cero en vez de usar elementos más elaborados y probados construidos por otros. Este hecho se deriva de la poca confianza que existe en los desarrollos de otras personas, y de los problemas de comunicación relativos al traspaso de *software*.
- **Comportamiento impredecible del *software*.-** El *software* puede verse como un sistema discreto[1], con multitud de variables que definen su estado en cada momento. Un pequeño cambio en una de esas variables puede llevar al sistema a un estado totalmente diferente. Esto puede verse como una alta sensibilidad al ruido, es decir, que un pequeño error (ruido) puede provocar un comportamiento totalmente erróneo.

1.1.1 Herramientas para tratar la complejidad

Existe una limitación en la capacidad humana para tratar la complejidad. Es por ello que para enfrentarnos a ella solemos utilizar técnicas de análisis como: descomponer, abstraer y jerarquizar.

Descomponer

Descomponer consiste en dividir sucesivamente un problema en problemas menores e independientes. Cuando se crean sistemas grandes es esencial la descomposición para poder abordar el problema. Tras la descomposición, cada una de las partes se trata y refina de manera independiente.

La descomposición reduce el riesgo de error cuando se construyen elementos grandes porque permite construir elementos de manera incremental utilizando elementos más pequeños en los que ya tenemos confianza.

Desde el punto de vista de la programación se puede hablar de dos formas de descomposición. La descomposición orientada a algoritmos y la descomposición orientada a objetos.

- **Descomposición algorítmica.-** El problema se descompone en tareas más simples. Luego, cada tarea se descompone a su vez otras más simples y así sucesivamente. Por ejemplo, es el enfoque utilizado al hacer la comida (primero prepararemos los alimentos y luego los coceremos; para prepararlos primero los limpiaremos y luego los trocearemos; para limpiarlos...).

[1] Básicamente un autómata de estados.

- **Descomposición orientada a objetos.–** El problema se descompone en objetos de cuya interacción surge la solución. Cada objeto a su vez se descompone en más objetos. Por ejemplo, es el enfoque utilizado al construir un coche (un coche funciona como resultado de la interacción del motor, las ruedas y el chasis; el motor funciona como resultado de la interacción de la batería, el carburador y los cilindros; la batería...).

La descomposición orientada a objetos tiene varias ventajas cuando se aplica a proyectos grandes. Para empezar, facilita una mayor reusabilidad de mecanismos comunes. Además, al permitir construir objetos que a su vez agrupan a otros objetos provee de una mayor economía de expresión. Se adapta mejor al cambio porque se basa en formas intermedias estables que se pueden utilizar de múltiples maneras.

Por otro lado la descomposición algorítmica aporta ventajas cuando los problemas a resolver son pequeños, ya que es muy sencillo el desarrollo de tareas concretas.

Abstraer

Para comprender un sistema muy complejo las personas solemos ignorar los detalles que nos parecen poco significativos y solemos concentrarnos en otros que consideramos esenciales, construyendo un modelo simplificado del sistema que se conoce como abstracción. Por ejemplo, cuando se desea representar en un mapa los accidentes geográficos de un país se ignora la división en provincias o regiones de ese país. Es decir, se hace abstracción de la división política de la superficie de ese país.

Jerarquizar

Esta técnica de análisis nos permite ordenar los elementos presentes en un sistema complejo. Ordenar los elementos en grupos nos permite descubrir semejanzas y diferencias que nos guían luego para comprender la complejidad del sistema. Por ejemplo, si nos piden enumerar las provincias Españolas las citaremos agrupándolas previamente por comunidades autónomas.

1.1.2 Evolución de los lenguajes de programación

El desarrollo de sistemas de *software* cada vez más grandes y complicados ha influido en la creación de lenguajes de programación que faciliten técnicas para tratar la complejidad. Esto a su vez ha permitido abordar problemas más complejos, para entrar en una espiral de evolución que nos trae hasta la actualidad.

La primera generación de lenguajes de programación aparece entre los años 1954 y 1958. En aquella época aparecieron *Fortran* (*Formula Translator* de la mano de J. W. Backus), *ALGOL*, *FlowMatic* e *IPL V*. Esta primera etapa se caracteriza por programas muy lineales, con una sola línea principal de ejecución, y por una clara orientación hacia ingenieros y científicos. En estos lenguajes no existe una separación clara entre datos y programas, y como mecanismo de reutilización de código se propone la biblioteca de funciones.

En la segunda generación (1959 – 1961) aparecen *Fortran II* y *Cobol* (*Common Business Oriented Language*). Este último tuvo un alto arraigo en el mundo empresarial al que iba dirigido. Además, aparece *Lisp* (*List Processing* creado en el *MIT* por J. McCarthy y otros) orientado a los problemas de inteligencia artificial y que tuvo un gran impacto en multitud de lenguajes posteriores. En todos estos lenguajes, aunque ya existe separación entre datos y programa, el acceso a los datos es desordenado y no se proporcionan mecanismos que preserven la integridad de los mismos. Utilizando estos lenguajes no existe ninguna restricción en cuanto al orden de los datos y además se puede acceder a ellos directamente desde cualquier parte de un programa. Así, los cambios en una parte de un programa, que acceda a ciertos datos, pueden ocasionar errores en otra parte del programa ya cerrada, que accede a los mismos datos. Lógicamente, esto da lugar a que los programas de cierta complejidad desarrollados con estos lenguajes sean muy inestables y difícilmente puedan crecer.

La tercera generación (1962 – 1975) es muy prolífica. Entre la multitud de lenguajes que aparecen podemos destacar *Pascal*, *C* y *Simula*. En esta etapa aparecen conceptos como el de programación estructurada y el de abstracción de datos. La programación estructurada se basa en un teorema de Dijkstra que demuestra que cualquier programa de ordenador puede escribirse con un lenguaje que permita la ejecución secuencial de instrucciones, la instrucción condicional y la realización de bucles de instrucciones. Por otro lado, las abstracciones de datos consisten en la definición de tipos complejos de datos y su asociación a operadores para tratarlos. Estas abstracciones permiten que se puedan abordar programas más complejos. Sin embargo, estos lenguajes aún no formalizan mecanismos de protección adecuados para evitar violaciones en los protocolos de acceso a los datos. Tampoco añaden ningún mecanismo de reutilización de código distinto a las bibliotecas de funciones.

En la etapa que comprende desde el año 1976 hasta 1980 los lenguajes de la etapa anterior evolucionan y se estandarizan. Aparecen además lenguajes funcionales como los de la familia *ML*, y lenguajes lógicos como *Prolog*.

En las décadas de 1980 y 1990 aparecen los lenguajes orientados a objetos como *SmallTalk*, *C++* y *Java*. Estos lenguajes están especialmente diseñados para adaptarse a la descomposición orientada a objetos. Para ello, existen mecanismos que permiten restringir el acceso a los datos que forman parte de los objetos. Es responsabilidad de cada objeto el mantenimiento de sus datos, y el resto de objetos que interaccionan con él lo hace a través de una interfaz bien definida. También aparece en estos lenguajes el mecanismo de herencia que permite la reutilización de código de una manera controlada.

Casi todos los lenguajes evolucionan desde sus orígenes. Por ejemplo, las versiones más actuales de *Cobol*, *Fortran* y *Pascal* incorporan características de orientación a objetos.

1.1.3 Paradigmas de programación

Un paradigma de programación es un modelo conceptual para desarrollar programas. El uso de un paradigma se refuerza por el lenguaje que se escoja para realizar un programa concreto, aunque en general, con mayor o menor dificultad, se puede usar cualquier lenguaje de programación para seguir cualquier paradigma. Grady Booch cita diferentes paradigmas:

- **Orientado al procedimiento.-** Que se expresa de manera imperativa en forma de algoritmos. Por ejemplo *C* y *Fortran*.
- **Orientado a funciones.-** Se basa en el concepto matemático de función y se expresa de manera declarativa. Por ejemplo *Lisp*, *SML*, *Hope*, *Hasckel*.
- **Orientado a la lógica.-** Que se expresa por metas en forma de cálculo de predicados. Utilizan reglas e inferencia lógica. Por ejemplo *Prolog*.
- **Orientado a objetos.-** Se expresa en forma de relaciones entre objetos y responsabilidades de cada uno. Por ejemplo *SmallTalk* o *Java*.

Esta clasificación no es estanca, algunos lenguajes pertenecen a varios paradigmas. Existen paradigmas más generales, como el **paradigma imperativo** o el **estructurado**, que engloban a otros. Además, continuamente aparecen nuevos paradigmas, como la **orientación a aspectos** que complementa a la Programación Orientada a Objetos fomentando la separación de conceptos.

Cada uno de estos paradigmas tiene ciertas ventajas. Así, el paradigma procedimental aporta ventajas cuando son tareas sencillas que se pueden describir con unos pocos pasos, o cuando es importante optimizar la velocidad de ejecución. El uso del paradigma orientado a la lógica facilita la implementación de sistemas expertos en los que se deba manejar una base de conocimiento. El paradigma funcional permite construir programas concisos, fáciles de probar y paralelizar, lo cual es muy adecuado para la prueba matemática de algoritmos y para programas en los que se requiera un alto grado de fiabilidad.

El paradigma de Programación Orientada a Objetos está demostrando su utilidad en una amplia variedad de problemas (interfaces gráficas, simuladores, aplicaciones ofimáticas, juegos...). Además, está demostrando que puede ser un marco de alto nivel ideal para integrar sistemas desarrollados siguiendo diferentes paradigmas.

1.2. Programación orientada a objetos

Antes de hablar de Programación Orientada a Objetos definiremos qué es un objeto. Un **objeto** es algo a lo que se le puede enviar mensajes y que puede responder a los mismos y que tiene un **estado**, un **comportamiento** bien definido y una **identidad**. El estado de un objeto está definido por el valor de ciertas variables internas al objeto. Este estado puede cambiar dependiendo de los mensajes que reciba desde el exterior o de un cambio interno al propio objeto. El comportamiento de un objeto varía en función del estado en el que se encuentra, y se percibe por los valores que devuelve ante los mensajes que recibe y por los cambios que produce en los objetos con los que se relaciona. Finalmente, la identidad de un objeto es aquello que lo hace distinguible de otros objetos.

La Programación Orientada a Objetos es un método de desarrollo en el cual los programas se organizan como colecciones de objetos que cooperan para resolver un problema. En general los objetos pueden corresponderse a entidades del mundo real (como un coche o un gato), a acciones (como saltar o realizar una transacción bancaria) o a procesos (como el vuelo o el aprendizaje).

La Programación Orientada a Objetos se basa en el **Modelo de Objetos**. Este modelo se fundamenta en el uso de 7 capacidades, 4 de las cuales que se consideran principales y 3 secundarias. Los lenguajes de programación orientados a objetos se caracterizan porque proporcionan mecanismos que dan soporte a estas capacidades.

Las capacidades principales son:

- Abstraer.
- Encapsular.
- Modularizar.
- Jerarquizar.

Las capacidades secundarias son:

- Tipo.
- Concurrencia.

- Persistencia.

A lo largo de las explicaciones que siguen y a lo largo del resto de temas se utilizará el **Lenguaje Unificado de Modelado** (*Unified Modeling Language* - ***UML***) para representar gráficamente los ejemplos que se vayan proponiendo. *UML* es un lenguaje gráfico para visualizar, especificar, construir y documentar los artefactos de un sistema *software* orientado a objetos. La notación *UML* se irá introduciendo de manera gradual según se vaya necesitando.

En los siguientes apartados se describen más detalladamente las capacidades principales antes mencionadas.

1.2.1 Abstraer

Los humanos hemos desarrollado la capacidad de abstracción para tratar la complejidad. Al analizar un problema ignoramos gran parte de los detalles, y solo manejamos las ideas generales de un modelo simplificado de ese problema. Por ejemplo al estudiar cómo conducir un coche nos podemos centrar en sus instrumentos (el volante, las marchas...) y olvidarnos de aspectos mecánicos (como el diámetro de los pistones o la aleación usada en el cigüeñal).

Abstraer es la capacidad que permite distinguir aquellas características fundamentales de un objeto que lo hacen diferente del resto, y que proporcionan límites conceptuales bien definidos relativos a la perspectiva del que lo visualiza. La abstracción surge de reconocer las similitudes entre objetos, situaciones o procesos en el mundo real, y la decisión de concentrarse en esas similitudes e ignorar las diferencias. Así, una abstracción se focaliza sobre una posible vista, ayudando a separar el comportamiento esencial de un objeto de su implementación. Volviendo al ejemplo de los mapas, un mapa político se focaliza en la división en regiones de la superficie de un país; si el mapa es físico se focaliza en los aspectos geográficos de la misma superficie.

Los lenguajes de programación orientados a objetos facilitan abstraer gracias a que permiten definir **interfaces** comunes para comunicarse con clases de objetos. Estas interfaces están compuestas por los **métodos**, que son funciones que pueden aplicarse sobre el objeto y que pueden verse como los mensajes que es posible enviar al objeto. Normalmente un objeto puede tener varias interfaces. En la Figura 1 se puede ver cómo el coche tiene distintas interfaces para comunicarse con los objetos que lo forman. Por ejemplo, el volante, la palanca de cambios o los pedales.

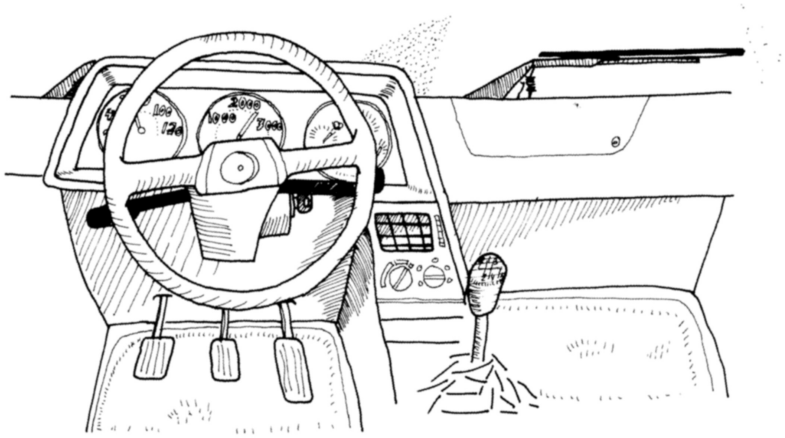


Figura 1.- Las interfaces nos permiten comunicarnos con los objetos, nos permiten concentrarnos en ciertos aspectos y obviar el resto.

Así, en algunos lenguajes de programación orientados a objetos (como *C++* o *Java*) aparece el concepto de **clase de objetos** o simplemente **clase** que une las interfaces definidas en el proceso de abstracción con la implementación del comportamiento deseado. Otros lenguajes (como *SmallTalk* o *JavaScript*) no permiten definir clases de objetos, sino que se basan en el concepto de prototipos (todo objeto es un prototipo a partir del cual se puede crear otro).

Las clases añaden a la definición de métodos la implementación de los mismos. También añaden las **propiedades**, que son variables internas al objeto o a la clase que definen el estado del objeto. Así, los objetos que hay en un sistema siempre pertenecen a una determinada clase, la cual define su comportamiento, la forma de interaccionar con él y sus posibles estados. Métodos y propiedades también se conocen como **miembros**.

Entre los objetos se crea un comportamiento **cliente/servidor**, de forma que el cliente conoce el comportamiento de una abstracción servidora analizando los servicios que presta. Estos servicios forman un **contrato** que establece las responsabilidades de un objeto respecto a las operaciones que puede realizar. El orden en que se deben aplicar las operaciones se conoce como **protocolo**, y pueden implicar **precondiciones** y **poscondiciones** que deben ser satisfechas. Cuando al enviar un mensaje a un objeto se cumplen las precondiciones pero el objeto no puede cumplir las poscondiciones se produce una **excepción**. Los lenguajes orientados a objetos también suelen dar soporte al manejo de excepciones.

Al estudiar un problema solemos fijar un nivel de abstracción, y en él buscamos las abstracciones presentes. No es frecuente mezclar elementos que están a diferente nivel de abstracción al analizar un problema debido a que genera confusión.

Por ejemplo, la estructura de un coche presenta varias abstracciones a un nivel alto de abstracción: motor, ruedas, chasis, puertas, volante. Dentro del motor, a diferente nivel de abstracción encontramos: cilindros, carburador, radiador, etc. A un nivel de abstracción menor, dentro de un carburador encontramos: tornillos, muelles, arandelas...

Otro ejemplo de niveles de abstracción podemos encontrarlo en la estructura de las plantas. A un nivel alto de abstracción están la raíz, el tallo y las hojas. Se aprecia que estos elementos tienen formas de interactuar bien definidas, y que no hay partes centrales que coordinen su funcionamiento. De la interacción entre estas abstracciones surge un comportamiento que es mayor que el comportamiento de sus partes.

A la hora de definir abstracciones es importante determinar la granularidad de los objetos, es decir, determinar qué son objetos y qué son partes de un objeto. Para ayudar a crear abstracciones se puede medir la calidad de una abstracción revisando los siguientes aspectos:

- **Acoplamiento**.- Minimizar el grado de asociación entre diferentes abstracciones.
- **Cohesión**.- Maximizar el grado de asociación dentro de una abstracción.
- **Suficiencia** y **completitud**.- Que tenga las características precisas para permitir un funcionamiento eficiente y completo.
- **Primitividad**.- Las operaciones de una abstracción deben ser lo más básicas posibles.

Sin embargo, debe tenerse en cuenta que encontrar buenas abstracciones en un problema no es una ciencia exacta.

Notación *UML*

En *UML* las clases se representan en los **Diagramas Estáticos de Clases** mediante rectángulos (ver Figura 4). En el interior de cada rectángulo se indican, separados por líneas horizontales, el nombre de la clase, las propiedades y los métodos.

Las interfaces se representan mediante los mismos rectángulos, con la salvedad de que en el nombre se antepone la palabra interfaz, y de que no se deja lugar para las propiedades ya que no tienen (ver Figura 2).

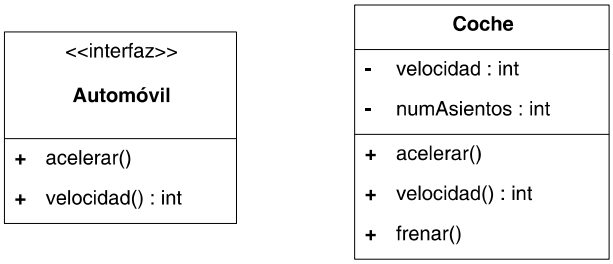


Figura 2.- Ejemplo de la clase `Coche` y de la interfaz `Automóvil`.

1.2.2 Encapsular

Encapsular es la capacidad que permite mantener oculta la implementación de una abstracción para los usuarios de la misma. El objetivo de encapsular es la **ocultación** de la implementación, para que ninguna parte de un sistema complejo dependa de cómo se ha implementado otra parte.

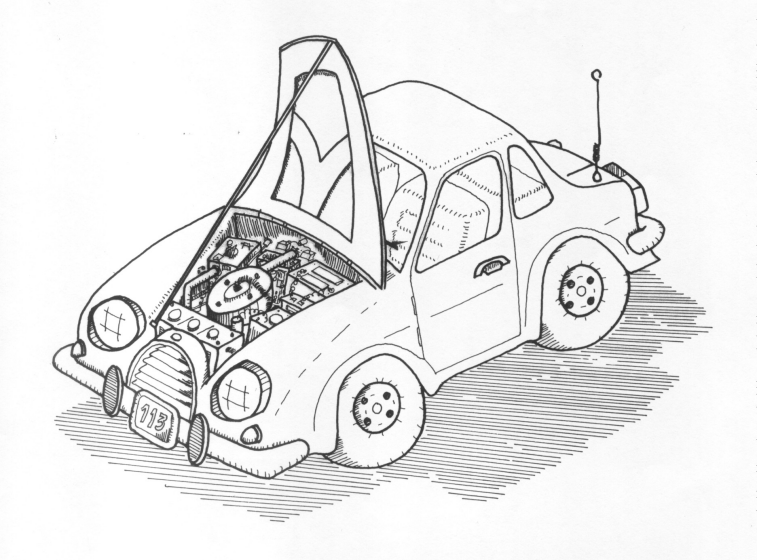


Figura 3.- La encapsulación oculta los detalles de implementación de una abstracción.

La abstracción y la encapsulación son complementarias. Mientras que la primera se centra en el comportamiento observable, la segunda lo hace en cómo se construye ese comportamiento. Así, la abstracción define la interfaz y la encapsulación se encarga de los detalles de la implementación. Para forzar esta útil separación entre abstracción e implementación, entre la definición de la interfaz y la implementación de una abstracción se dice que existe la **barrera de la abstracción**.

La principal ventaja de la encapsulación está en que facilita que al cambiar el funcionamiento interno de una abstracción, los clientes de la misma no lo noten. Para facilitar la ocultación los lenguajes orientados a objetos ofrecen ciertos mecanismos, como los modificadores de visibilidad ***public***, ***private*** y ***protected*** de *C++* y *Java*. Estos modificadores permiten que se pueda acceder libremente a unas partes de la interfaz (que son públicas), acceder con restricciones a otras partes (que son protegidas), y que se prohíba el acceso a otras partes (que son privadas).

Notación *UML*

Los Diagramas Estáticos de Clases se pueden adornar con los modificadores de visibilidad. Éstos se disponen precediendo a los métodos y a las propiedades, mediante los caracteres + , - y # para público, privado y protegido respectivamente.

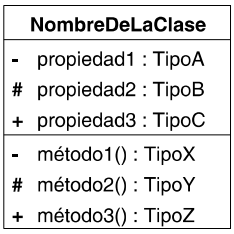


Figura 4.- Representación de una clase en *UML*.

1.2.3 Jerarquizar

En cualquier problema simple se encuentran más abstracciones de las que una persona puede usar a la vez en un razonamiento. Los conjuntos de abstracciones a menudo forman jerarquías y su identificación permite simplificar la comprensión del problema.

Jerarquizar es una capacidad que permite ordenar abstracciones. Su principal ventaja consiste en que la organización de las abstracciones de un sistema en una jerarquía permite detectar estructuras y comportamientos comunes y con ello simplificar el desarrollo.

En el esquema de Programación Orientada a Objetos se definen dos formas básicas de jerarquías:

- Jerarquías entre clases e interfaces.
- Jerarquías entre objetos.

Jerarquía entre clases e interfaces

Definen relaciones del tipo "tal abstracción es una abstracción cual". Estas relaciones se denominan relaciones de herencia y cumplen que los elementos de los que se hereda son más generales, mientras que los que elementos que heredan están más especializados.

Un ejemplo lo podríamos encontrar en la clase `Coche` y en una clase derivada `Deportivo`. Los objetos de la clase `Deportivo` compartirían el comportamiento de los de la clase `Coche`, pero además los objetos de la clase `Deportivo` añadirían ciertas interfaces y comportamientos nuevos. La clase `Deportivo` sería una especialización de la clase `Coche`. Por ejemplo, los objetos de la clase `Deportivo` podrían tener la propiedad `turbo`, pero un `Coche` en general no tendría por qué tener `turbo`. Además, la clase `Deportivo` podría redefinir el método `acelerar`, para dotarlo de diferentes características mediante una implementación diferente. Obsérvese que todo `Deportivo` sería un `Coche`, pero no todo `Coche` sería un `Deportivo`.

Los lenguajes de programación orientados a objetos deben dar soporte a la jerarquía de clases e interfaces. Para ello deben proporcionar mecanismos para definir clases e interfaces nuevas o que hereden de otras ya existentes. Por eso se puede hablar de herencia entre interfaces, de herencia entre clase e interfaz y de herencia entre clases.

- Herencia entre interfaces.- Ya se ha dicho que las interfaces no contienen implementación ni propiedades, sólo definen métodos. Por eso, cuando una interfaz hereda de otra lo que obtiene es precisamente la declaración de sus métodos.
- Herencia entre clase e interfaz.- Cuando una clase hereda de una interfaz se dice que tal clase implementa o que cumple tal interfaz. Nuevamente, como las interfaces no contienen implementación sólo se hereda la definición de métodos, siendo responsabilidad de la clase que hereda implementar el comportamiento.
- Herencia entre clases.- La herencia de clases se produce cuando una clase hereda tanto la interfaz como el comportamiento de otra clase. Por ejemplo, cuando en *Java* se define una clase `B` que hereda de otra `A`, la clase `B` tiene todos los métodos y todas las propiedades que en `A` se definieron como públicas o protegidas. Además, cualquier llamada a un método `M` de la clase `B` se comportará exactamente como lo haría sobre un objeto de la clase `A`, salvo que explícitamente se modifique su comportamiento en `B`.

Tras heredar siempre se pueden añadir nuevos métodos a la interfaz que hereda para especializarla en algún aspecto. Además, en el caso de que la que hereda sea una clase, y no una interfaz, también se pueden añadir nuevas propiedades.

Por otro lado, la herencia puede clasificarse en dos grupos atendiendo a la forma de las jerarquías: herencia simple y herencia múltiple.

- **Herencia simple**.- Cuando un elemento sólo hereda de una jerarquía. En la práctica esto ocurre cuando una clase o interfaz hereda sólo de otra y no lo hace de varias simultáneamente.
- **Herencia múltiple**.- Cuando una clase o interfaz hereda simultáneamente de varias jerarquías diferentes.

La herencia múltiple de clases da lugar a lo que se conoce como el **problema de la ambigüedad**. Este problema aparece cuando una clase hereda de varias que tienen un método de idéntico nombre y parámetros pero que tiene definidos comportamientos diferentes en cada clase.

El ejemplo habitual de ambigüedad se conoce como el **problema del diamante** (ver Figura 5). En él hay una clase `A`, de la que heredan dos clases `B` y `C`. Finalmente existe una clase `D` que hereda simultáneamente de `B` y `C`. Algunos de los interrogantes que plantea esta situación son:

- Si existe un método en `A` que se implementa de forma diferente en `B` y en `C`, ¿a qué método se invoca cuando se hace desde `D`?
- Cuando se invoca al constructor de `D` y este a su vez invoca recursivamente a los constructores de las clases de las que deriva: ¿cuántas veces se invoca al constructor de la clase `A`, y en qué orden se invocan los constructores de `B` y `C`?

Las respuestas a estas preguntas resultan en esquemas de ejecución complicados que pueden sorprender fácilmente al programador. Por otro lado, la herencia múltiple de interfaz no presenta problemas de ambigüedad, pues la implementación sólo está detallada en las clases y su uso resulta menos complejo aunque la reutilización de código es menor.

Es por todo esto que en algunos lenguajes (como *Java*) se ha decidido no dar soporte a la herencia múltiple de clases.

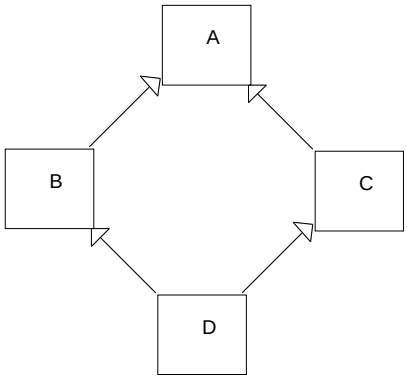


Figura 5.- Problema del diamante.

Herencia	Simple	Múltiple
Entre interfaces	Sí	Sí
Entre clase e interfaz	Sí	Sí
Entre clases	Sí	No

Figura 6.- Relaciones de herencia soportadas por *Java*.

En cierto sentido la herencia de clases puede comprometer la encapsulación, pues para heredar se deben conocer detalles de la clase padre. Este hecho crea una complejidad que normalmente debe evitarse. Por ello se recomienda en general el uso exclusivo de la herencia de interfaz frente a otros tipos de herencia.

Jerarquía entre objetos

Las jerarquías entre objetos se pueden clasificar en 2 tipos de relaciones: **relaciones de asociación** y **relaciones de dependencia**.

Las relaciones de asociación establecen relaciones estructurales entre objetos de forma que se establece una conexión entre ellos. Fundamentalmente, este tipo de relaciones permite construir objetos mediante la asociación de otros objetos menores. Un ejemplo lo podemos encontrar en la relación entre los objetos de la clase `Coche` y los objetos de la clase `Rueda`, si definimos que un objeto de la clase `Coche` posee cuatro objetos de la clase `Rueda`.

Los lenguajes orientados a objetos facilitan las relaciones de asociación permitiendo que cualquier clase pueda se pueda utilizar para definir una propiedad dentro otra clase.

Las relaciones de dependencia dan lugar a relaciones del tipo "tal objeto usa tal otro objeto" por lo que también se conocen como **relaciones de uso**. Estas relaciones se distinguen de las de asociación porque el ámbito y el tiempo de uso de un objeto desde otro es más limitado.

Los lenguajes orientados a objetos facilitan las relaciones de dependencia permitiendo que un método pueda utilizar un objeto de manera local.

Notación *UML*

En *UML* las relaciones entre clases se representan en los Diagramas Estáticos de Clases mediante líneas que unen las clases.

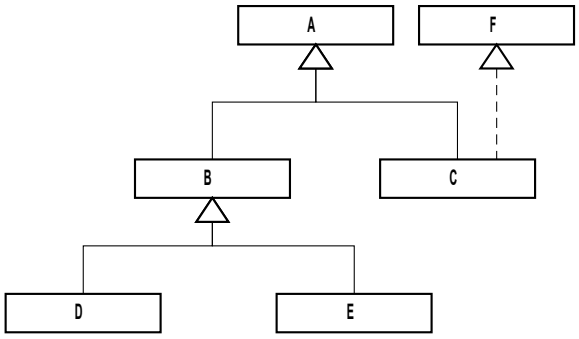


Figura 7.- Las clases `D` y `E` derivan de la `B`. A su vez `B` y `C` derivan de `A`. La clase `C` además implementa la interfaz `F`.

En el caso de la herencia la relación se representa mediante una línea entre las cajas correspondientes a las clases involucradas en la relación. Dicha línea termina en una flecha hueca que apunta a la clase de la que se hereda. Si la herencia es de interfaz, la línea es punteada (ver Figura 7).

Si la clase `B` hereda de `A` se dice que `A` es la **superclase** de `B`. También se suele decir que `A` es la **clase padre** de `B`, que `B` es una **clase derivada** de `A`, o que `B` es una **clase hija** de `A`. Además, se habla de que una clase está a un nivel de abstracción mayor cuanto más alta está en una jerarquía de clases. Así, la clase más general en una jerarquía se conoce como **clase base**. Por ejemplo, en la Figura 7 `B` esta en un nivel de abstracción mayor que `D` y `E`, siendo base de ellas, y la clase `A` es base de `B`, `C`, `D` y `E`.

Por otro lado, las relaciones de asociación entre objetos se representan mediante líneas simples que unen las cajas de las clases. Sobre estas líneas se pueden indicar las **cardinalidades**, es decir, las proporciones en las que intervienen los elementos de la asociación (ver Figuras 8 y 9).

Obsérvese que las relaciones de asociación también se pueden representar como propiedades. Suele hacerse así cuando el diseño de una de las clases de la asociación no tiene importancia en ese contexto. En estos casos, el nombre del objeto menor se añade como una propiedad de la clase que se está explicando. Por ejemplo, en la clase `Coche` de la Figura 9, existe una asociación entre la clase `Coche` y la clase `Texto`, pero se considera de menos importancia que la asociación entre `Coche` y `Rueda`, quizás porque la clase `Texto` es muy común, y por ello se presenta de manera abreviada.

Finalmente, las relaciones de uso se representan mediante líneas punteadas con punta de flecha, indicando el sentido de la dependencia.

Figura 8.- Esta figura presenta que cada objeto de la clase `A` puede estar asociado a varios elementos de la clase *B*. Y cada elemento de la clase `B` sólo está asociado a un elemento de la clase `A`.

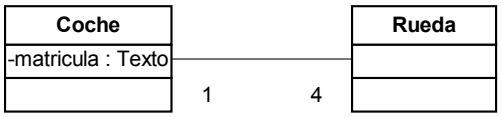


Figura 9.- Ejemplo en el que los objetos de la clase `Coche` se relacionan con 4 elementos de la clase `Rueda` y con uno de la clase `Texto`, aunque a esta última asociación se le concede menor importancia. Además, los elementos de la clase `Rueda` se asocian con uno de la clase `Coche`.

Figura 10.- La clase A usa a la clase B.

También se suelen utilizar diagramas que presentan instancias de objetos mediante cajas y las relaciones que hay entre ellos en un momento de la ejecución mediante líneas. Estos diagramas se denominan **Diagramas de Instancias**. En el interior de las cajas se presenta el nombre de la clase precedida por el carácter de dos puntos. Opcionalmente se puede presentar el nombre de la instancia, en cuyo caso debe aparecer subrayado y antes del nombre de la clase. Finalmente, bajo el nombre de la clase puede presentarse el valor de la propiedades de la clase para esa instancia.

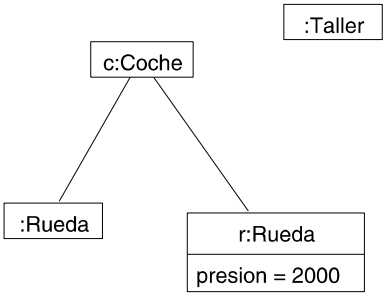


Figura 11.- Diagrama de Instancias.

1.2.4 Modularizar

Modularizar es la capacidad que permite dividir un programa en agrupaciones lógicas de sentencias. A estas agrupaciones se las llama **módulos**.

Las ventajas que ofrece la modularidad son:

- Facilidad de mantenimiento, diseño y revisión. Al dividir el programa se facilita que varias personas pueden desarrollar de manera simultánea e independiente conjuntos disjuntos de módulos.
- Aumento de la velocidad de compilación. Los compiladores suelen compilar por módulos. Esto significa que el cambio de un módulo sólo implica la recompilación del módulo y de los que dependan de él, pero no la del total de módulos.
- Mejora en la organización y en la reusabilidad, ya que es más fácil localizar las abstracciones similares si se encuentran agrupadas de una manera lógica.

A la hora de diseñar los módulos debe tenerse en cuenta:

- Maximizar la **coherencia**, es decir, se deben agrupar en un mismo módulo las abstracciones relacionadas lógicamente.
- Minimizar las **dependencias** entre módulos, es decir, que para compilar un módulo no se necesite compilar muchos otros.
- Controlar el **tamaño de los módulos**. Módulos pequeños aumentan la desorganización, módulos muy grandes son menos manejables y aumentan los tiempos de compilación.

En *C++* y en *Java* el concepto de módulo encuentra soporte a varios niveles. Al menor nivel cada módulo se corresponde a un fichero. Así, los ficheros se pueden escribir y compilar de manera separada. Las bibliotecas aportan un segundo nivel de modularidad a *C++*. Mientras, en lenguajes como *Java*, se ha creado el concepto de **paquete** que permite un número ilimitado de niveles de modularidad. También suele utilizarse el concepto de componente y de programa como módulos que tienen una funcionalidad completa e independiente.

Notación *UML*

En *UML* los paquetes se representan en los **Diagramas de Paquetes** mediante unos rectángulos que se asemejan a carpetas. Estas carpetas se etiquetan con el nombre del paquete.

Los componentes y los programas se representan utilizando unas cajas decoradas con dos cajas en su interior (ver Figura 12). Obsérvese que entre paquetes se pueden dar relaciones de asociación y dependencia.

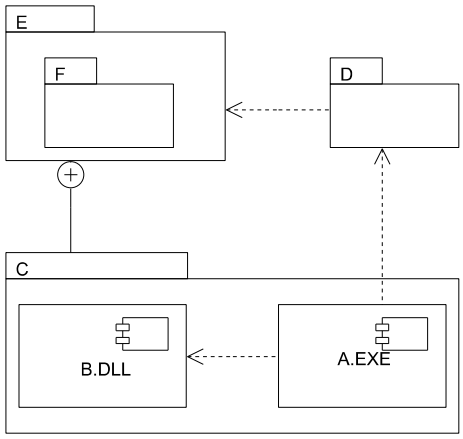


Figura 12.- El paquete `D` depende del paquete `E`, y el componente `A.EXE` depende del componente `B.DLL` y del paquete `D`. Por otro lado, los componentes `A.EXE` y `B.DLL` están contenidos en el paquete `C` que igual que `F` está contenido en el paquete `E`.

1.2.5 Tipo

Un tipo[2] es una caracterización precisa asociada a un conjunto de objetos. En Programación Orientada a Objetos, los objetos que comparten una misma interfaz se dice que tienen el mismo tipo. También se dice que el tipo de un objeto `B` deriva del de otro `A` cuando la interfaz de `B` es un superconjunto de la de `A`.

La asociación del tipo a un objeto se conoce como **tipado**. El tipado refuerza las decisiones de diseño, impidiendo que se confundan abstracciones diferentes y dificultando que puedan utilizarse abstracciones de maneras no previstas.

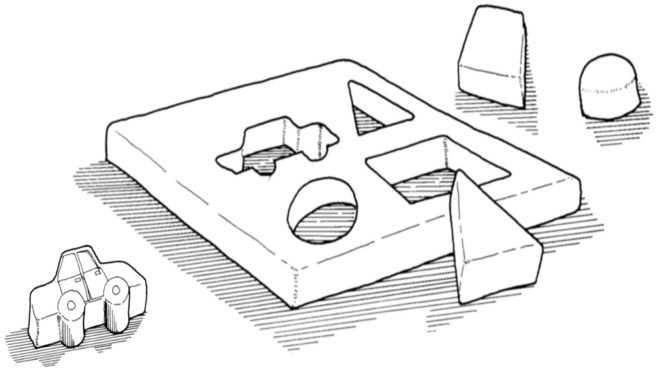


Figura 13.- El tipado protege de los errores que se pueden cometer al mezclar abstracciones.

El soporte al tipado en los diferentes lenguajes de programación es desigual. Por ejemplo, en *SmallTalk* los mensajes fluyen de un objeto a otro sin ninguna restricción, aunque los objetos puedan no responder al recibir mensajes para los que no estén preparados.

Los lenguajes pueden clasificarse en dos grupos respecto a las restricciones que impone el tipo:

- Lenguajes con **tipado fuerte**, en los que no es posible mezclar variables de tipos diferentes.
- Lenguajes con **tipado débil**, en los que es posible mezclar variables de diferentes tipos. Esta mezcla puede realizarse de manera implícita o explícita mediante **coerción**. La coerción fuerza al compilador a tratar un dato de un tipo como si fuese de otro tipo, y por ello el programador debe responsabilizarse de tal violación.

No se debe confundir la coerción con la **conversión** o ***casting***. La coerción trata al dato como un simple conjunto de *bytes* forzando su uso como de un tipo determinado, mientras que la conversión implica algún tipo de tratamiento que asegura la validez de la operación.

El tipado fuerte evita los errores que se pueden cometer al mezclar abstracciones de una manera no prevista en el diseño. Por ejemplo, puede evitar que se comparen directamente dos importes si están en monedas diferentes (como dólares y euros). También acelera los intérpretes y los compiladores, pues al ser más estricta la sintaxis el número de posibilidades a analizar es menor.

[2] El concepto de tipo proviene directamente de la Teoría de Tipos Abstractos. Según ésta un Tipo Abstracto de Datos es la estructura resultante de la unión de un dominio para ciertos datos y de una colección de operaciones que actúan sobre esos datos.

En contra del tipado fuerte se puede decir que introduce fuertes dependencias semánticas entre las abstracciones. Esto puede provocar que un pequeño cambio en la interfaz de una clase base desencadene que tenga que recompilarse todas las subclases y clases relacionadas.

En *Java* el tipado es fuerte. Tanto una clase como una interfaz definen un tipo, y con él los mensajes que se le pueden enviar. Así, todos los objetos de un determinado tipo pueden recibir los mismos mensajes. Además, *Java* impide violar el tipo de un objeto enviándole mensajes para los que no está preparado.

Existen lenguajes que tienen un tipado híbrido entre tipado fuerte y débil. En este grupo se encuentra por ejemplo *C* y *C++*, que tiene tipado fuerte en algunos casos (por ejemplo al comparar clases), pero débil en otros (al permitir comparar tipos básicos como el `int` y el `float` o al usar coerción de punteros a `void`).

Por otro lado, dependiendo de la forma en que se declaran las variables se puede hablar de:

- **Tipado explicito**, cuando antes de utilizar una variable debe declararse el tipo al que pertenece.
- **Tipado implícito**, cuando las variables no se les indica el tipo, sino que éste se deduce del código.

El tipado implícito puede ocasionar problemas difíciles de detectar debido a errores de escritura. Si en un lenguaje con tipado implícito se define la variable "entero" y luego se usa la variable "enetero"[3] se crearán dos variables diferentes sin que se produzca ningún error sintáctico. Sin embargo, en ejecución estará asegurado el error en la semántica del código. Para evitar estos problemas *Java* utiliza tipado explicito.

Por último, en cuanto al momento en que se comprueba el tipo de una variable, los lenguajes de programación se pueden clasificar en:

- Lenguajes con **tipado estático**, en los que el tipo se comprueba en compilación.
- Lenguajes con **tipado dinámico**, en los que el tipo se comprueba en ejecución.

En general, los lenguajes no utilizan sólo tipado estático o sólo tipado dinámico. Pues aunque un lenguaje pretenda utilizar sólo tipado estático, puede haber ciertas características que obligan a utilizar el tipado dinámico.

Así, los lenguajes compilados suelen utilizar tipado estático (por ejemplo *C++* y *Java*). En estos lenguajes, tras compilar llega la etapa de enlazado. Cuando el tipo de un objeto se conoce en compilación las direcciones de los métodos se enlazan mediante lo que se conoce como un **enlace temprano**. Sin embargo, en los lenguajes de programación orientados a objetos existe una característica derivada de la herencia que impide el enlace temprano en algunas ocasiones. Esta característica se denomina **polimorfismo** y consiste en permitir utilizar una misma variable para designar objetos de clases diferentes pero que cumplan la misma interfaz. El polimorfismo puede impedir que se sepa en compilación a qué método de qué clase se debe llamar. Por lo que se debe realizar la comprobación de tipo en ejecución y realizar un **enlace tardío**. Cuando en compilación se puede determinar la dirección del método se habla de **polimorfismo estático**, y cuando sólo es posible determinarla en ejecución se habla de **polimorfismo dinámico**.

Otro ejemplo lo podemos encontrar en lenguajes como *C++* y *Java* en los que se realiza coerción con comprobación de tipos en tiempo de ejecución[4].

Se debe señalar que cuando el tipado es implícito y dinámico, el tipo de una variable podría cambiar durante la ejecución, como por ejemplo permite el lenguaje *Python*. En este lenguaje también se considera una variedad del tipado dinámico que se denomina **tipado latente** y que consiste en que al invocar un método de un objeto no se comprueba el tipo como algo global sino que sólo se comprueba la existencia del método que se está invocando.

El siguiente diagrama compara la forma en la que se utiliza el tipado en algunos lenguajes de programación.

[3]Obsérvese que intencionadamente sobra un carácter "e".

[4]En *C++* se pueden realizar conversiones y coerciones. Utilizando la instrucción `dinamic_cast` para realizar la coerción con comprobación de tipos dinámica; utilizando `static_cast` para coerción con comprobación de tipos estática; con `reinterpret_cast` para coerción sin comprobación de tipos; finalmente usando paréntesis y * se realiza coerción, mientras que usando solo paréntesis se realiza conversión.

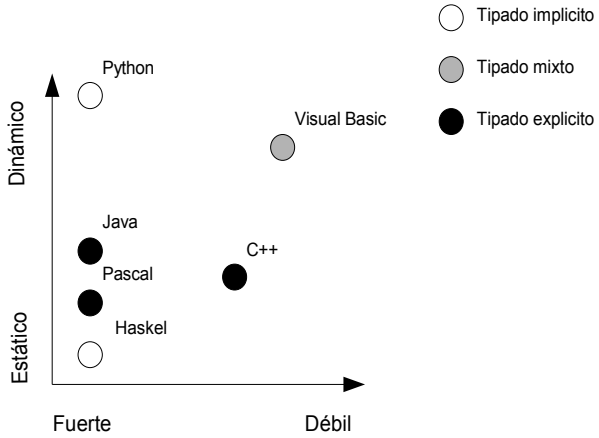


Figura 14.- Comparación del tipado de varios lenguajes.

Notación *UML*

En *UML* el tipo aparece como nombre de cada clase o interfaz. También suele seguir a cualquier identificador que aparezca en un diagrama, separado del mismo por el símbolo de dos puntos (ver Figuras 2 y 9).

1.2.6 Concurrencia

La concurrencia es la capacidad que permite la ejecución paralela de varias secuencias de instrucciones. Hay problemas que se resuelven más fácilmente si se dispone de esta capacidad. Por ejemplo, hay ocasiones en las que se dispone de múltiples procesadores y se desea aprovecharlos a todos a la vez para resolver un mismo problema. Otros problemas requieren que se puedan tratar diversos eventos simultáneamente.

Clásicamente los lenguajes de programación no han dado ningún soporte a la concurrencia. Generalmente, esta facilidad es proporcionada por los sistemas operativos. Por ejemplo, en *Unix* la concurrencia se consigue con la invocación de una función del sistema operativo llamada `fork` que divide la línea de ejecución, creando múltiples **líneas de ejecución** (también conocidas como **hilos** o ***threads***).

Los lenguajes orientados a objetos pueden dar soporte a la concurrencia de una manera natural haciendo que un objeto se pueda ejecutar en un *thread* separado. A tales objetos se les llama objetos activos frente a los pasivos que no se ejecutan en *threads* separados. Esta forma de tratamiento ayuda a ocultar la concurrencia a altos niveles de abstracción. Sin embargo, los problemas clásicos de la concurrencia persisten (condiciones de carrera, *deadlock*, inanición, injusticia, retraso...). *Java* es un ejemplo de lenguaje que da soporte a la concurrencia creando hilos al crear ciertos objetos, aunque en el caso de *Java* el hilo puede ejecutar luego código que esté en otros objetos.

1.2.7 Persistencia

La persistencia es la capacidad que permite que la existencia de los datos trascienda en el tiempo y en el espacio.

Podemos clasificar los datos en relación a su vida según los siguientes 6 tipos:

- Expresiones. Cuya vida no supera el ámbito de una línea de código.
- Variables locales. Cuya vida se circunscribe a la vida de una función.
- Variables globales. Que existen mientras se ejecuta un programa.
- Datos que persisten de una ejecución a otra.
- Datos que sobreviven a una versión de un programa.
- Datos que sobreviven cuando ya no existen los programas, los sistemas operativos e incluso los ordenadores en los que fueron creados.

Los tres primeros puntos entran dentro del soporte dado clásicamente por los lenguajes de programación. Los tres últimos puntos no suelen estar soportados por los lenguajes de programación, entrando en el ámbito de las bases de datos.

Un lenguaje orientado a objetos que dé soporte para la persistencia debería permitir grabar los objetos que existan, así como la definición de sus clases, de manera que puedan cargarse más adelante sin ambigüedad, incluso en otro programa distinto al que lo ha creado. *Java* da cierto nivel de soporte a la persistencia de una clase si ésta cumple la interfaz predefinida `Serializable`. Esta interfaz define métodos que permiten almacenar los objetos en soportes permanentes.

1.3. Lecturas recomendadas

"*Object-Oriented Analysis and Design*", G. Booch, Benjamin Cummings, 1994. Los cinco primeros capítulos de este libro constituyen una de las primeras, y mejores, introducciones formales al diseño orientado a objetos de la mano de uno de los creadores de *UML*.

"El lenguaje Unificado de Modelado" y "El lenguaje Unificado de Modelado: Guía de referencia", G. Booch, I. Jacobson y J. Rumbaugh, 2ª edición, Addison Wesley 2006. "Los tres amigos", creadores de *UML*, presentan en estas dos obras una guía de usuario y una guía de referencia de *UML*. Imprescindibles para conocer todos los detalles de *UML,* su carácter, poco didáctico, hace que solo se recomiende como obra de consulta.

1.4. Ejercicios

Ejercicio 1

Supóngase que un banco desea instalar cajeros automáticos para que sus clientes puedan sacar e ingresar dinero mediante una tarjeta de débito.

Cada cliente podrá tener más de una cuenta en el Banco, y por cada cuenta se podrá tener como máximo una tarjeta; de cada cuenta sólo interesan los datos del titular de la misma, estando las tarjetas, en caso de que existan, a nombre del titular de la cuenta.

Para evitar que las tarjetas extraviadas o robadas se usen, se decide que antes de entregar el dinero del cajero debe verificar mediante una contraseña la identidad del propietario de la cuenta. Sólo se permiten tres intentos para introducir la clave correcta, si no se consigue se invalida la tarjeta.

Para aumentar la seguridad, el Banco propone que se fije una cantidad máxima de dinero que pueda sacarse cada día.

El banco desea que más tarde sea posible añadir nuevas operaciones al cajero como: consultar el saldo, comprar entradas de teatro, etc.

Usando el paradigma de orientación a objetos y el lenguaje *UML* se pide construir un Diagrama Estático de Clases que pueda representar una solución al enunciado.

Capítulo 2 Introducción al lenguaje Java

2.1. Introducción

En este capítulo se presenta una breve introducción al lenguaje *Java*. La bibliografía recomendada al final del capítulo permitirá comprender con detalle los conceptos que en los siguientes puntos se esbozan.

2.1.1 Breve historia de *Java*

First Person, una filial de *Sun Microsystems* especializada en el desarrollo de *software* para pequeños dispositivos, decidió desarrollar un nuevo lenguaje adecuado a sus necesidades. Entre estas necesidades estaban la reducción del coste de pruebas en relación a otros lenguajes como *C* o *C++*, la orientación a objetos, la inclusión de bibliotecas gráficas y la independencia del sistema operativo. Así, de la mano de James Gosling, nació *Oak*. Al poco tiempo, en 1994, cerró *First Person* al no despegar ni los proyectos de TV interactiva, ni los de tostadoras inteligentes.

Uno de los desarrolladores de *Unix* y fundadores de *Sun*, Bill Joy, pensó que *Oak* podía ser el lenguaje que la incipiente Internet necesitaba y en 1995, tras una pequeña adaptación de *Oak*, nacía *Java*.

Entre las principales características de *Java* se pueden citar:

- Sintaxis similar a la de *C++*. Aunque se simplifican algunas características del lenguaje como: la sobrecarga de operadores, la herencia múltiple, el paso por referencia de parámetros, la gestión de punteros, la liberación de memoria y las instrucciones de precompilación.
- Soporte homogéneo a la Programación Orientada a Objetos. A diferencia de *C++*, que puede considerarse un lenguaje multiparadigma, *Java* está diseñado específicamente para utilizar el paradigma de orientación a objetos.
- Independencia de la plataforma. Con *Java* se hizo un importante esfuerzo para que el mismo código fuese ejecutable independientemente del *hardware* y del Sistema Operativo.

A la versión del lenguaje vigente en 2009 se le denomina *Java 5* y supone una mejora sustancial respecto a aquella primera versión de *Java* de 1995. El siguiente diagrama de bloques muestra un esquema de alto nivel de la plataforma *Java SE*.

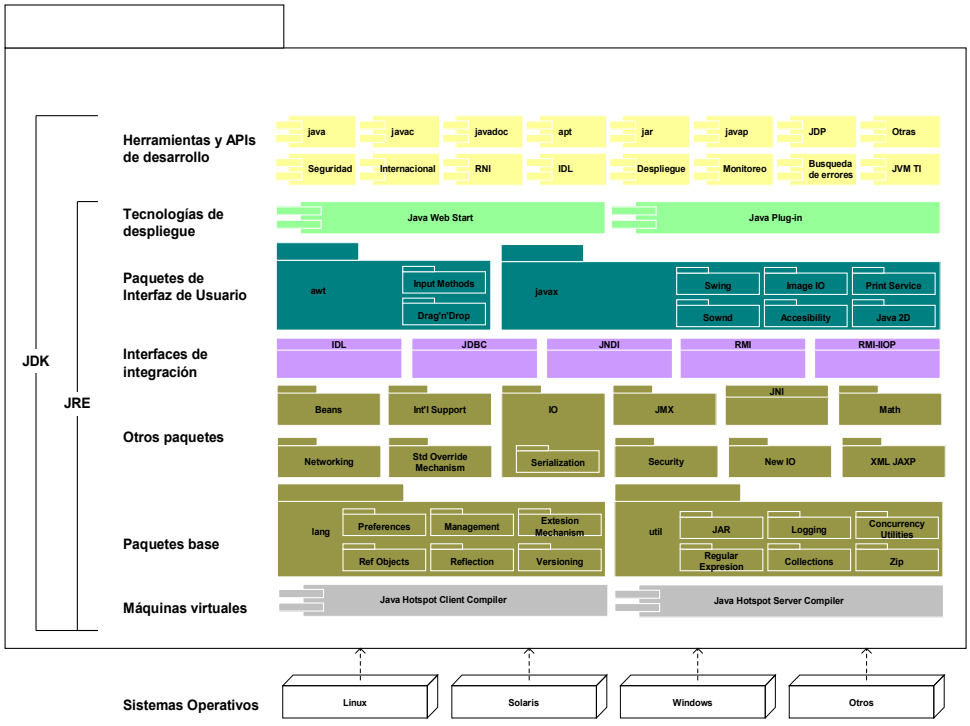


Figura 15.- Diagrama de los principales componentes de la plataforma *Java SE*.

2.1.2 La máquina virtual de *Java*

Partiendo del **código fuente** de un programa, el compilador de *Java SE* no produce **código máquina** ejecutable en un procesador específico. En vez de esto, genera un código simplificado, denominado ***bytecode***, que precisa de otro programa, la **Máquina Virtual** de *Java SE,* para ejecutarse. Esta Máquina Virtual, ejecuta las aplicaciones *Java* en un entorno virtual que se denomina la **caja de arena** (***sandbox***). Este esquema de funcionamiento hace que *Java* sea muy seguro en cuanto al acceso no autorizado a los recursos del sistema. También hace que sea multiplataforma, pues una sola compilación genera *bytecodes* que se podrán ejecutar en máquinas virtuales de diferentes plataformas.

Debido a la existencia de este lenguaje intermedio de *bytecodes*, hay quien considera a *Java* un lenguaje interpretado. Sin embargo, en *Java SE* desde la versión 1.3 la Máquina Virtual en vez de interpretar cada *bytecode*, compila a código máquina nativo cada función antes de ejecutarla. Este tipo de enfoque se denomina ***Just In Time*** (***JIT***) porque se realiza la compilación en el momento de la ejecución (esta misma tecnología la ha copiado *Microsoft* en la máquina virtual de la plataforma *.NET*). Además, los *bytecodes* tienen un formato similar al código máquina de cualquier procesador, y por ello la generación de código máquina desde estos *bytecodes* es muy rápida.

Debe existir una Máquina Virtual diferente para cada plataforma que soporte *Java SE*. Incluso suele haber diferentes máquinas virtuales para la misma plataforma, dependiendo del fabricante de *software*, siendo algunas mejores que otras.

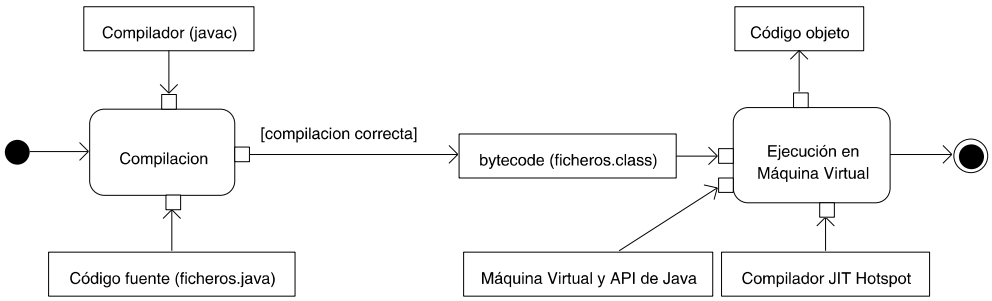


Figura 16.- Esquema básico de compilación y ejecución en *Java*.

También existen implementaciones del compilador de *Java* que convierten el código fuente directamente en código objeto nativo, como *GCJ*. Esto elimina la etapa intermedia donde se genera el *bytecode*, pero la salida de este tipo de compiladores no es multiplataforma.

2.1.3 Compiladores de *Java*

Existen varias aplicaciones para compilar *Java* en línea de comandos. La más simple es *javac*, proporcionada por el *JDK* (*Java Developer Kit*) de *Sun*. Actualmente, en 2009, la última versión estable del kit de desarrollo de *Java* de *Sun* es el *JDK* 6.

2.1.4 Entornos de desarrollo para *Java*

Para programar en *Java* existen entornos *Open Source* de nivel profesional como *Eclipse* (desarrollado inicialmente por *IBM*) y *Netbeans* (de *Sun*) que hacen esta tecnología abierta y accesible. También existen entornos privativos como *IntelliJ IDEA*. Todos estos entornos permiten programación de aplicaciones de tipo consola, aplicaciones visuales, aplicaciones *web*, para sistemas embebidos, para dispositivos móviles...

Tanto *Eclipse*, como *Netbeans* e *IntelliJ IDEA*, están desarrollados en *Java.* También hay entornos más modestos como *Kawa* o *Scite*, desarrollados completamente en lenguaje nativo, que son menos pesados en ejecución pero que no contemplan muchas de las características que aportan sus hermanos mayores.

2.1.5 El programa hola mundo

La tradición, iniciada con el famoso libro sobre el lenguaje *C* de Kernighan y Ritchie, dicta que el primer programa que debe ejecutarse cuando se está aprendiendo un lenguaje es el programa `HolaMundo`. Dicho programa se limita a imprimir en pantalla un mensaje de bienvenida. Para no romper la tradición, con nuestro editor de texto *ASCII* favorito, creamos un fichero fuente denominado `HolaMundo.java`. En él copiamos el siguiente fragmento de código, respetando mayúsculas y minúsculas, tanto en el nombre del fichero como en el contenido, pues *Java* presta atención a este detalle.

```
/* Programa en Java que muestra un mensaje de bienvenida */
class HolaMundo {

   // método en el que comienza la ejecución del programa
   public static void main (String [ ] arg)     {
       System.out.println("Hola mundo");
    }
}
```

Una vez grabado el código fuente del programa en un directorio a nuestra elección, desde la línea de comandos invocamos al compilador.

```
c:\swa\Ejemplo\> javac HolaMundo.java
```

Éste compila y genera el fichero `HolaMundo.class` que contiene los *bytecodes*. Finalmente ejecutamos el programa invocando a la máquina virtual de *Java*, obteniendo el saludo esperado.

```
c:\swa\Ejemplo\> java -classpath . HolaMundo
Hola mundo
```

2.1.6 Los comentarios

Los comentarios son líneas de texto que aparecen en el código fuente y que son obviadas por el compilador. Su principal utilidad consiste en ayudar a entender el código fuente adyacente.

Dentro de un programa *Java* hay dos formas de escribir un comentario. La primera, que llamaremos comentario en bloque, se hereda de *C* y utiliza una combinación del carácter barra inclinada hacia la derecha seguido del carácter asterisco para abrir el comentario, y la combinación asterisco barra inclinada para cerrarlo. La otra forma se hereda de *C++* y simplifica la tarea de hacer comentarios de una sola línea. Para ello utiliza dos veces el carácter de la barra inclinada a la derecha para indicar el comienzo del comentario y el retorno de carro para terminarlo. Puede observarse que, por definición, no es posible anidar comentarios de tipo *C*, aunque sí es posible que un comentario de tipo *C* incluya comentarios de línea.

```
/*
   Esto es un comentario que puede ocupar
   varias líneas.
*/

// Esto es un comentario de línea
```

Además, *Java* aconseja un modo estándar de escritura de comentarios que, de seguirse, permite al programa `javadoc` del *JDK* de *Sun* generar de manera automática la documentación del código. El programa `javadoc` produce un archivo *HTML* por cada fichero que analiza que puede visualizarse con cualquier navegador *web*. Los comentarios de documentación son comentarios de bloque que repiten el asterisco. Los comentarios de documentación se deben colocar antes del bloque de definición al que hacen referencia, porque *Java* asocia el comentario de documentación a la primera sentencia que se encuentra después del propio comentario.

En los comentarios de documentación se utilizan ciertas palabras clave precedidas del carácter arroba para etiquetar los elementos que se describen. La siguiente tabla recoge las diferentes posibilidades.

`@param`	Parámetros de entrada para un método
`@return`	Valor devuelto por un método
`@throws`	Excepciones lanzadas por un método
`@version`	Versión del código
`@author`	Autor del código
`@deprecated`	El método ya no se debe usar, pero se mantiene por compatibilidad
`@see`	Referencias a otras clases
`@since`	Nueva funcionalidad añadida en la versión que marque este tag

Tabla 1.- Anotaciones utilizadas en los comentarios.

El siguiente fragmento de código muestra un ejemplo de comentarios de documentación. Además, en todos los fragmentos de código que se utilicen en este libro se incluirá el *javadoc* correspondiente como parte del ejemplo.

```
/** Ejemplo de documentación automática con javadoc
   @author José Vélez
   @versión 1.0
*/

/**
*  Descripción del comportamiento del método.
*  @param nombreParámetro_1 descripción
*  @param nombreParámetro_n descripción
```

```
*  @return   descripción
*  @throws   nombre_clase descripción
*  @since 0.6
*  @deprecated explicación
*/
```

2.1.7 Juego de instrucciones

En el siguiente párrafo se presentan las palabras reservadas del lenguaje *Java*.

```
abstract, boolean, break, byte, case, catch, char, class, continue, default, do, double, extends,
false, final, finally, float, for, if, implements, instanceof, int, interface, long, native, new,
null, package, private, public, return, short, static, super, switch, synchronized, this,
threadsafe, transient, true, try, void, volatile, while.
```

2.2. Expresiones básicas y control de flujo

En *Java* existen una serie de tipos de datos básicos predefinidos que se conocen como **tipos primitivos**. Además, es posible declarar **variables** de estos tipos primitivos y asignarlas a datos del mismo tipo mediante el operador de asignación y una expresión adecuada.

En adelante se utilizará un metalenguaje denominado *GOLD* para describir la sintaxis de *Java*. Este metalenguaje presenta los símbolos no terminales entre ángulos(<>), los símbolos terminales como cadenas libres, los corchetes para indicar elementos opcionales y elementos de las expresiones regulares (como el asterisco para indicar repeticiones). El uso de estas expresiones no tienen el ánimo de ser completas respecto a la sintaxis de *Java*, sino de aportar precisión al uso más habitual de dicha sintaxis.

Por ejemplo, la declaración e inicialización de una variable tiene el siguiente formato.

```
<declaración>      ::=     [final] <tipo> <identificador> [,<identificador>];
<inicialización>   ::=     <identificador> = <expresión de inicialización>;
```

Obsérvese que es posible indicar que el valor de la variable será **inmutable** (no cambiará tras su primera inicialización) mediante el modificador `final`. Además, las operaciones de declaración e inicialización se pueden realizar en un solo paso.

```
<declaración e inicialización> ::= [final] <tipo> <identificador> [= <expresión de inicialización >];
```

Los identificadores en *Java* pueden corresponder a cualquier cadena de caracteres *Unicode* siempre que no comiencen por un número o un símbolo utilizado por *Java* para los operadores, ni coincida con una palabra reservada de *Java*. Así, se puede definir:

```
int mainCont, auxCont;     //Declaración de dos enteros
mainCont = 2;              //Inicialización de un entero
int i = 5;                 //Declara e inicia un entero
final double π = 3.14159;  //Declara e inicia una constante real
```

Para mejorar la comunicación es mejor utilizar identificadores significativos y, a ser posible, en inglés[5]. Respecto al uso de mayúsculas, en *Java* suele seguirse el convenio del **camello** al nombrar las clases e interfaces. Este convenio consiste en comenzar cada palabra que forme parte del identificador en mayúsculas y utilizar minúsculas para el resto de los caracteres, dando a los identificadores una silueta con jorobas. Las propiedades, métodos y variables utilizan el criterio del camello excepto para la primera letra del identificador. Las constantes se escriben en mayúsculas, separando las palabras con la barra horizontal baja. Finalmente, los paquetes se suelen escribir en minúsculas.

Por último se debe destacar que la declaración de las variables se puede realizar en cualquier punto del código. Esto debe aprovecharse para declarar las variables en el punto en que se utilizan y no antes, mejorando con ello la claridad del código.

2.2.1 Tipos primitivos

Los **tipos primitivos** en *Java* se pueden agrupar en cuatro categorías: números enteros, números reales, caracteres y booleanos. Cada uno tiene un valor por defecto que sólo se aplica cuando la variable corresponde a una propiedad de una clase.

[5] Usar inglés al programar facilita la comunicación entre programadores de distintas nacionalidades. Por ejemplo, para un castellano hablante, leer código en chino puede ser tan dificultoso, como para que un habitante de Pekín lea código en castellano. Por otro lado, ciertas extensiones de *Java* obligan a utilizar partículas como `get` o `put` las cuales junto a palabras en castellano presentan un aspecto, cuanto menos, singular (por ejemplo `getValor`). No obstante, este libro, destinado a castellano hablantes, el código estará en castellano.

Enteros

Existen diferentes tipos para representar números enteros. Cada uno permite un rango diferente de representación y como consecuencia también tiene un coste de memoria diferente.

Tipo	*Bytes*	Descripción	Valor por defecto	Sintaxis	Ejemplo de uso
`byte`	1	Enteros desde –128 a 127	`0`	-{0,1}[0-9]$^{+}$	`byte b = 100;`
`short`	2	Enteros desde –16384 hasta 16383	`0`	-{0,1}[0-9]$^{+}$	`short s = -8000;`
`int`	4	Enteros desde -2^{31} hasta 2^{31}-1	`0`	-{0,1}[0-9]$^{+}$	`int i = 1400000;`
`long`	8	Enteros desde -2^{63} hasta 2^{63}-1	`0`	-{0,1}[0-9]$^{+}$	`long l = -53;`

Reales

Hay dos tipos para representar reales. De nuevo cada uno tiene una precisión, un rango de representación y una ocupación de memoria diferentes.

Tipo	*Bytes*	Descripción	Valor por defecto	Sintaxis	Ejemplo de uso
`float`	8	Reales	`+0.0f`	-{0,1}[0-9]*\.[0-9]$^{+}$f	`float e = 2.71f;` `float x = -1.21f;`
`double`	16	Reales largos	`0.0`	-{0,1}[0-9]*\.[0-9]$^{+}$	`double p = +3.14;`

Lógicos

Las expresiones lógicas están soportadas por el tipo bivaluado `boolean`.

Tipo	*Bytes*	Descripción	Valor por defecto	Sintaxis	Ejemplo de uso
`boolean`	indefinido	cierto o falso	`false`	true | false	`boolean c = true;`

Caracteres

El tipo primitivo para los caracteres es el tipo `char`.

Tipo	*Bytes*	Descripción	Valor por defecto	Sintaxis	Ejemplo de uso
`char`	2	Caracteres *Unicode*	`'\u000000'`	'[\u[0-9]$^{+}$|.]'	`char c = 'ñ';` `char D = '\u13';`

Por herencia de C la barra inclinada hacia la izquierda se utiliza como carácter de control. Por separado no tiene sentido, sino que siempre se combina con el siguiente carácter. Algunos de sus principales usos son:

- Si va seguida de una `u` y un número su significado es el carácter representado por el valor *Unicode* de ese número.
- Si va seguida de otra barra igual o de comillas dobles equivale al carácter barra o el de comillas respectivamente.
- Si va seguida de una `n` significa retorno de carro.
- Si va seguida de una `t` significa tabulado.

2.2.2 Conversiones entre tipos primitivos

Es posible realizar conversiones directas entre los diferentes tipos básicos numéricos siempre que se realice de menor precisión a mayor precisión. Es decir si se sigue el siguiente flujo:

```
byte > short > int > long > float > double
```

Para cualquier conversión en otra dirección se debe utilizar la operación de *casting*. Consiste en poner el tipo al que se desea convertir por delante y entre paréntesis. Las conversiones por *casting* deben hacerse de manera explícita para evitar

equívocos, pues pueden provocar pérdida de precisión (o de todo el valor). El siguiente ejemplo muestra la conversión de un valor de tipo flotante a un valor de tipo *byte* usando un *casting*.

```
float temperatura = 25.2f;
byte temperaturaTruncada = (byte) temperatura; //valdrá 25
```

2.2.3 Operadores

Los **operadores** de *Java* se pueden agrupar en: relacionales, aritméticos, lógicos y de bit.

Relacionales

Operador	Descripción	Ejemplo
==	Igualdad	`a == b`
>	Mayor	`a > b`
<	Menor	`a < b`
>=	Mayor o igual	`a >= b`
<=	Menor o igual	`a <= b`
!=	Distinto	`a != b`

Aritméticos (sobre enteros y reales)

Operador	Detalle	Unitario	Binario	Ejemplo
+	Suma o declaración de positivo	X	X	`a + b, +a`
-	Resta o declaración de negativo	X	X	`a - b, -a`
*	Producto		X	`a * b`
/	División		X	`a / b`
%	Resto de la división de dos números enteros		X	`a % b`
++	Post o pre-incremento en una unidad de una variable	X		`a++, ++a`
--	Post o pre-decremento en una unidad de una variable	X		`a--, --a`
+=	Incremento en varias unidades de una variable		X	`a += b`
-=	Decremento en varias unidades de una variable		X	`a -= 5`
*=	Multiplicación por un número de una variable		X	`a *=3`
/=	División por un número de una variable		X	`a /= b`
%=	Modularización por un número de una variable		X	`A %=4`

Lógicos

Operador	Descripción	Unitario	Binario	Ejemplo
`&`	Conjunción (and)		X	`a&b`
`|`	Disyunción (or)		X	`a|b`
`&&`	Conjunción impaciente (and)		X	`a&&b`
`||`	Disyunción impaciente (or)		X	`a||b`
`^`	Disyunción exclusiva (xor)		X	`a^b`
`!`	Negación (not)	X		`¡a`
`&=`	Asignación con conjunción		X	`a&=b`
`|=`	Asignación con disyunción		X	`a|=b`

Bit (sobre tipos enteros)

Operador	Descripción	Unitario	Binario	Ejemplo
`&`	Conjunción (and)		X	`a&b`
`|`	Disyunción (or)		X	`a|b`
`^`	Disyunción exclusiva (xor)		X	`a^b`
`<<`	Desplazamiento binario a la izquierda rellenando con ceros		X	`a<<3`
`>>`	Desplazamiento binario a la derecha rellenando con el bit más significativo		X	`a>>3`
`>>>`	Desplazamiento binario a la derecha rellenando con ceros		X	`a>>>3`
`~`	Negación binaria	X		`~a`

2.2.4 Control del flujo

Java dispone de la mayoría de sentencias de control de flujo habituales en otros lenguajes de programación como: `if`, `while`, `for`... En los siguientes puntos se analizan estas sentencias y los ámbitos sobre los que se aplican.

Definición de ámbitos y sentencias

Un ámbito se inicia con el carácter de llave abierta hacia la derecha y se termina con el carácter de llave abierta hacia al izquierda. *Java* utiliza los ámbitos para definir las sentencias afectadas por una declaración o por una instrucción de control de flujo. Los ámbitos se pueden anidar y dentro de un ámbito siempre puede definirse otro. Esto, unido a que las declaraciones de las variables son locales al ámbito en que se declaran, hace que el ámbito también se utilice para agrupar lógicamente sentencias.

Las sentencias del lenguaje son las responsables de declarar métodos, propiedades y variables, crear objetos, invocar funciones de los objetos, controlar el flujo de ejecución, etc. Todas las sentencias terminan en punto y coma salvo las que se aplican sobre la sentencia o ámbito inmediato. Por ello, las sentencias pueden escribirse unas tras otras en la misma línea, aunque normalmente suele disponerse una sola sentencia por línea para mejorar la legibilidad.

El siguiente ejemplo ilustra los conceptos que se acaban de exponer.

```
{
  int a; //Aquí a no tiene valor definido
  a = 25; int c = 15; //Aquí 'a' vale 25 y 'c' vale 15
  {
    int b = 2; //Aquí 'a' vale 25 y 'b' vale 2
    //Java no permite declarar dos identificadores iguales en un ámbito
    //por lo que en este punto no sería posible escribir int c = 12;
  }
  //Aquí 'a' vale 25, 'c' vale 15 y 'b' no está declarada
}
```

Instrucciones `if – else`

La sentencia `if` lleva asociada una expresión booleana entre paréntesis. De cumplirse la expresión se ejecuta la sentencia o el ámbito siguiente al `if`. En otro caso se ejecuta, si existe, la rama `else`.

```
<instrucción if-else> ::= if (<expression booleana>)<ámbito> | <sentencia>
                          [else <ámbito> | <sentencia>]
```

El siguiente ejemplo ilustra el uso de `if` encadenado a otro `if`.

```
if (importe>100) {
   descuento = 0.2;
}
else if (importe<80) {
   descuento = 0.1;
}
else {
   descuento = 0.15;
}
```

Instrucciones `for`

La palabra reservada `for` permite repetir una sentencia o un ámbito cualquier número de veces. Su estructura es la siguiente.

```
<instrucción for> ::=
  for (<expresión inicialización>;<expresión booleana>;<expresión incremento>)
    <ámbito> | <sentencia>
```

Esta instrucción en primer lugar ejecuta la expresión de inicialización. Luego evalúa la expresión booleana y en caso de resultar verdadera hace una ejecución del ámbito. Al terminar ejecuta la expresión de incremento. El proceso de evaluación de la expresión, ejecución del ámbito y posterior incremento, se repite hasta que la expresión booleana deja de resultar cierta.

Los bucles `for` suelen utilizarse cuando el número de iteraciones es concreto y no varía. El siguiente ejemplo ilustra su uso:

```
for (int cont = 0; cont < 100; cont++) {
   suma += cont;
}
```

Instrucciones `while`

La palabra reservada `while` también permite repetir un ámbito cualquier número de veces. Suele utilizarse cuando el número de iteraciones no es concreto y depende de varias condiciones. Su estructura es la siguiente.

```
<instrucción while> ::= while (<expresión booleana>) <ámbito> | <sentencia>
```

Su funcionamiento consiste en evaluar la condición y en caso de cumplirse hacer una ejecución del ámbito. Este proceso se repite hasta que la condición deja de cumplirse. Un ejemplo de su uso sería:

```
while ((cont > 100) && (!terminar)) {
   terminar = (cont > minimoVariable);      //terminar toma valor booleano
   cont-=2;                                 //se resta 2 a cont cada iteración
}
```

Instrucciones `do while`

`Do-While` es similar a `while` pero obliga a una primera iteración antes de evaluar la condición.

```
<instrucción do> ::= do <ámbito> | <sentencia> while (<expresión booleana>);
```

Un ejemplo de su uso sería:

```
do {
    calculo *= 25;          //calculo se multiplica por 25 en cada iteración
    cont>>1;                //cont se divide por dos cada iteración
} while ((cont > 0) && (calculo <1000));
```

Instrucciones `break y continue`

La instrucción `break` permite interrumpir en cualquier punto la ejecución normal de un bucle `for` o `while` y salir instantáneamente del mismo.

La instrucción `continue` permite interrumpir la ejecución normal de un bucle `for` o `while` y volver a la sentencia de evaluación para decidir si continuar o salir.

Tanto `break` como `continue` pueden utilizarse en combinación con etiquetas para salir de varios bucles simultáneamente hasta alcanzar el bucle etiquetado. Las etiquetas se definen mediante un identificador seguido del carácter de dos puntos.

Estas instrucciones no se corresponden con ninguna de las imprescindibles para la programación estructurada. Sin embargo, muchos programadores (entre ellos los diseñadores de lenguajes) consideran que la posibilidad de salir de un bloque de código en cualquier punto, facilita la tarea de programar y no trae ningún inconveniente ya que es fácilmente reproducible con una estructura condicional adecuada.

Instrucciones `switch-case-break-default`

Permite evaluar una sola vez una expresión aritmética y en base a su resultado ejecutar las sentencias de un ámbito concreto. Su estructura es la siguiente:

```
<instrucción switch> ::= switch (<expresión entera>) '{'
    [case <valor_entero> ':' '{' <sentencias> [break';'] '}' ]*
    [default             ':' '{' <sentencias>          '}' ]     '}'
```

2.3. Las clases

Las clases son el mecanismo básico que proporciona *Java* para manejar el concepto de abstracción y de tipado (ver capítulo 1). *Java* permite construir clases que definan la interfaz y la implementación de los objetos que posteriormente se podrán crear. Así, cada clase define una interfaz y un tipo (o varios tipos en el caso de las clases parametrizadas).

Las clases en *Java* están constituidas por:

- **Identificación.-** La primera línea de una clase identifica el nombre de la clase, las clases de las que hereda, las interfaces que implementa, las excepciones que puede lanzar y los parámetros utilizados para referir tipos.
- **Miembros.-** Se pueden clasificar en datos miembros y funciones miembros, también conocidos como propiedades y métodos, de los que ya hablamos en el punto 1.2.1. Cabe decir que en *Java* tanto las propiedades como los métodos pueden corresponder a instancias de la clase (objetos) o a la propia clase (con valores y comportamientos comunes para todos los objetos de la clase).
- **Clases internas.-** Clases que se definen dentro de otras clases. Normalmente se utilizan para crear clases fuertemente ligadas con la clase huésped. Estas clases internas pueden incluso ser anónimas, derivando de otra que le proporciona una interfaz con el resto del código.
- **Bloques de inicialización.-** Conjuntos de instrucciones encargadas de iniciar las propiedades de la clase. *Java* se encarga de que estos bloques se ejecuten convenientemente antes de crear los objetos de la clase.

El siguiente esquema muestra cómo se define una clase en *Java*. Se aprecia que además de nombrar la clase con un identificador, se especifican las excepciones que puede lanzar la clase (de lo que hablaremos en el capítulo 4) y los parámetros utilizados para referenciar tipos genéricos (de lo que hablaremos en el capítulo 5). Además, al definir una clase también se puede añadir información sobre la herencia. Aunque se profundizará en este aspecto en el capítulo 3, se puede adelantar que para heredar de otra clase se utiliza la palabra reservada `extends` seguida del nombre de la clase.

```
<clase> ::= [Modificador de clase] class <identificador>
            [parámetros] [herencia] [excepciones]
'{'
   [<método>|<propiedad>|<inicializacion>|<clase>]*
'}'
```

De nuevo, el identificador puede estar formado por cualquier cadena de caracteres *Unicode* siempre que no comience por un número o un símbolo utilizado por *Java* para los operadores, ni coincida con una palabra reservada del lenguaje.

La definición de una clase en *Java* siempre se realiza dentro de un paquete. Para especificar el paquete al que pertenecen las clases definidas en un fichero se usa la palabra `package` seguida del nombre del paquete. Si no se especifica ningún nombre de paquete el fichero se incluye en un "paquete por defecto" correspondiente al directorio inicial de ejecución.

Las propiedades se declaran en el interior de una clase mediante la definición de variables. Por otro lado, los métodos se declaran mediante un identificador seguido de unos paréntesis, que pueden contener los parámetros pasados al método. En general, tras los parámetros suele existir un ámbito que contiene la implementación del método, en dicho ámbito se opera con los parámetros, con las propiedades y con los otros métodos de la clase utilizando simplemente sus identificadores. El identificador del método debe estar precedido por el tipo del valor que devuelve o `void` si no devuelve nada, mientras que en el interior del método se usará la palabra reservada `return` para indicar el valor devuelto.

Los siguientes ejemplos definen la clase `Automóvil` y la clase `Coche`. Ambos tienen definidas algunas propiedades y métodos. Obsérvese que la clase `Coche` hereda de `Automóvil`, y por lo tanto, aunque no los declara explícitamente, tiene los miembros definidos en `Automóvil` más los que ella misma define.

```
/**
 * Ejemplo de implementación de una clase
 * @version 1.0
 */
class Automóvil {
    int velocidad;     //Declaración de una propiedad para la velocidad del coche

    // Ejemplo de declaración e implementación de un método
    /** Método que permite conocer la velocidad actual
     *  @return Entero con la velocidad
     */
    int velocidad() {
        //Implementación del método
        return velocidad;
    }
}

/**
 * Declaración de una clase que hereda de la clase Automovil
 * @version 1.0
 */
class Coche extends Automóvil {
    boolean enMarcha;          //Indica si el coche está en marcha
    int numRuedas = 4;         //Cuenta las ruedas del coche

    /** Método que permite aumentar la velocidad
     *  @param incremento Valor entero que se sumará a la velocidad
     */
    void acelerar(int incremento) {
        velocidad += incremento;
        enMarcha = true;
    }

    /**
     * Permite reducir la velocidad
     */
    void frenar() {
        if (enMarcha)
            velocidad --;
        if (velocidad == 0)
            enMarcha = false;
    }
}
```

En la Figura 17 se muestra un Diagrama Estático de Clases que representa las clases `Automóvil` y `Coche` y la relación que existe entre ellas.

Una vez que se tiene una clase se podrán crear objetos de esa clase y definir variables que referencien a tales objetos. La forma de declarar esas variables es similar a la ya descrita para los tipos primitivos.

```
    <declaración objeto> ::= [final] <tipo> <identificador>
                             [= new <nombre_clase>([parametros])];
```

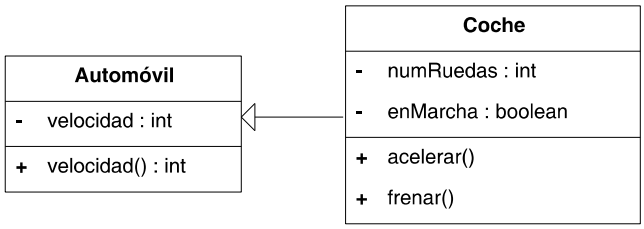


Figura 17.- Representación *UML* del ejemplo del `Automóvil` y el `Coche`.

Por ejemplo, para declarar una variable `miCoche` que referencie a objetos de la clase `Coche` deberemos escribir:

```
Coche miCoche;
```

Y para crear un objeto:

```
miCoche = new Coche();
```

También es posible declarar la referencia y crear el objeto en una única instrucción. Por ejemplo:

```
Coche miCoche = new Coche();
```

Una vez creado un objeto, podemos acceder a sus métodos y propiedades utilizando la variable que lo referencia y el identificador del miembro al que se desea acceder separado por un punto. Por ejemplo:

```
miCoche.velocidad = 15;
miCoche.acelerar(10);
```

2.3.1 Las propiedades en detalle

Las propiedades, o **campos**, sirven para dotar de estado al objeto o a la propia clase. Las propiedades son variables que se definen dentro de una clase y que pueden tomar valores diferentes en cada objeto. Estas variables corresponden a tipos primitivos o a referencias a objetos. Una referencia a un objeto es una variable cuyo contenido apunta a un objeto de una clase o a un identificador de referencia vacía (`null`).

Tipo	*Bytes*	Descripción	Valor por defecto	Ejemplo de uso
referencia a objeto	4	Referencia a un objeto que cumple un tipo	`null`	`Coche c = b;`

Al definir el identificador de estas variables se debe seguir las mismas reglas que para las variables de tipos primitivos. Es decir, el identificador puede estar formado por cualquier cadena de caracteres *Unicode* siempre que no comience por un número o un símbolo utilizado por *Java* para los operadores, ni coincida con una palabra reservada del lenguaje. Además, el identificador de la propiedad no puede coincidir con el nombre de la clase, ni en una clase puede haber dos propiedades con el mismo identificador.

La definición de una propiedad en una clase sigue la gramática adjunta. En caso de omitirse la inicialización la propiedad toma el valor por defecto.

```
<propiedad> ::= [mod control acceso] [mod uso] <tipo> <identifica> [= inicializacion];
```

Los modificadores de control de acceso más habituales son `public` y `private`, indicando si puede o no puede accederse a la propiedad desde fuera de la clase. Respecto a los modificadores de uso lo más habitual es no poner nada o poner `final` para indicar que su valor no cambia.

Como ya se ha dicho, se pueden distinguir dos tipos de propiedades: propiedades de los objetos y propiedades de la clase (también llamadas estáticas). Las primeras se pueden consultar en los objetos que se creen de esa clase y las segundas sobre la propia clase. Cuando la propiedad es estática se debe usar el modificador de uso `static`.

Acceso a las propiedades de objeto

Para acceder a las propiedades de objeto se utiliza el identificador del objeto seguido del identificador de la propiedad separados por un punto.

Así, para asignar el valor 4 a la propiedad `numRuedas` del objeto `miCoche` se podría hacer:

```
miCoche.numRuedas = 4;
```

De nuevo, si se desea acceder a una propiedad de un objeto desde dentro de un método no estático de la propia clase no es necesario hacer referencia al propio objeto, sino que se puede invocar al nombre de la propiedad directamente. Por ejemplo:

```
/**
* Declaración de la clase coche
* @version 2.0
*/
class Coche {
   public int numRuedas;

   /**
   * Añade ruedas a un coche
   * @deprecated Se ha decidido no mantener este método
   */
   public void añadirRuedas() {
       numRuedas++;
   }
}
```

Acceso a las propiedades de clase (propiedades estáticas)

Las variables estáticas son aquellas cuya existencia es independiente de que haya objetos de la clase a la que pertenecen. Para acceder, desde fuera de la clase, a las propiedades estáticas de una clase se utiliza el nombre de tal clase seguido de un punto y del nombre de la propiedad. Por ejemplo, sumar 1 a la propiedad `numeroUnidadesVendidas` de la clase `Coche`, que es común para todos los objetos de la clase, se podría hacer así:

```
Coche.numUnidadesVendidas++;
```

Como siempre, si se accede a una propiedad estática de una clase desde dentro del ámbito de la propia clase no es necesario hacer referencia a la clase.

```
/**
* Declaración de la clase coche
* @version 3.0
*/
class Coche {
   public static int numUnidadesVendidas = 0;

   /**
   * Aumenta el número de unidades vendidas
   */
   public void venderUnidad() {
       numUnidadesVendidas++;
   }

   //Bloque de inicialización estático
   static {
       numUnidades = 25;
   }
}
```

La referencia `this`

La propiedad `this` es una referencia no modificable al propio objeto en el que se invoca y, por lo tanto, se puede utilizar en todos los métodos no estáticos de un objeto. Permite pasar referencias del propio objeto en el que se invoca a otros objetos. También sirve para referirse a propiedades de la propia clase, resolviendo la ambigüedad que aparece cuando un método tiene parámetros o variables con nombres iguales a las propiedades de la clase. Por ejemplo:

```
/**
* Declaración de la clase coche
* @version 4.0
*/
class Coche {
   int velocidad;                    //declaración de una propiedad
```

```
    /**
    * Acelera el coche
    * @param velocidad Aumento en la velocidad
    */
    public void acelerar(int velocidad) { //Parámetro de igual nombre a propiedad
      this.velocidad += velocidad;      //Se suma el parámetro sobre la propiedad
    }
}
```

2.3.2 Los métodos en detalle

Los métodos (también llamados mensajes) son funciones definidas dentro la clase y que se invocan sobre los objetos creados de esa clase o sobre la clase misma.

Cada método consta de un identificador que nuevamente puede estar formado por cualquier cadena de caracteres *Unicode*, siempre que no comiencen por un número o un símbolo utilizado para los operadores, ni coincida con una palabra reservada. El siguiente cuadro muestra la gramática de definición de un método.

```
<método> ::= [Modificador de control de acceso] [Modifcador de uso] <tipo>
       <Identificador>([parámetros]) [excepciones] [{[sentencia]*}]

<parámetros> ::= <tipo> <identificador>[,<tipo> <identificador>]*
<excepciones> ::= throws <tipo> [,<tipo>]*
```

Obsérvese que opcionalmente los métodos pueden declarar el lanzamiento de excepciones mediante la palabra reservada `throws` seguida del tipo de excepción lanzada. Dicho tipo debe ser `Exception` o un derivado. A su vez, para lanzar una excepción en el cuerpo del método se utiliza la palabra reservada `throw` seguida del objeto lanzado. En el capítulo siguiente se tratará con más detalle del manejo de excepciones.

También en los métodos, los modificadores de control de acceso más habituales son `public` y `private`, para indicar si puede o no puede invocarse un método desde fuera de la clase. Respecto a los modificadores de uso, lo más habitual es utilizar `final` para indicar que su implementación no puede cambiarse mediante herencia o `abstract` para indicar que el método no tiene implementación y solo define una interfaz. Por otro lado, cuando el método es estático se debe usar el modificador de uso `static`. Este tipo de métodos no tiene acceso a las propiedades no estáticas.

Acceso a los métodos de un objeto

Para acceder a un método de un objeto se utiliza el identificador del objeto seguido del identificador del método. Ambos nombres se disponen separados por el carácter punto y van seguidos de unos paréntesis que pueden contener los parámetros que se le pasan al método. Por ejemplo:

```
miCoche.acelerar(10);
```

Si se desea acceder a un método de una clase desde otro método no estático de la propia clase se puede invocar al nombre del método directamente sin anteponer ningún nombre.

Acceso a los métodos de una clase (métodos estáticos)

Para acceder a los métodos estáticos de una clase se utiliza el nombre de la clase seguido del nombre del método separados por el carácter punto, siendo innecesario referir la clase si se invoca en el ámbito de la propia clase.

Devolución de valores

Los métodos en *Java* pueden devolver valores de tipos primitivos o referencias a objetos. Para ello se utiliza una sentencia que consiste en la palabra reservada `return` seguida opcionalmente de una expresión. Esta expresión tras evaluarse corresponderá a una referencia a un objeto o a un tipo primitivo.

Los métodos que declaran el tipo del valor devuelto con la palabra reservada `void` no devuelven nada. En estos métodos, el uso de `return` (sin expresiones a su derecha) no es obligatorio, pero puede utilizarse para terminar la ejecución del método en cualquier punto.

```
<salida> ::= return [<expresión>];
```

Paso de parámetros

El paso de parámetros a un método en *Java* siempre es por valor. Tanto los tipos primitivos como las referencias a los objetos se pasan por valor. Así, un cambio de uno de estos parámetros dentro de un método no afecta a su valor en el exterior.

Sin embargo, en el caso de referencias a objetos debe tenerse en cuenta que el objeto al que referencian estos parámetros sí es el original. Por lo tanto cualquier cambio en el estado del objeto tiene reflejo fuera del método.

Los parámetros que se pasan a un método también admiten el modificador `final` para indicar que no es posible cambiar su valor.

Sobrecarga de métodos (polimorfismo estático)

Se pueden definir varios métodos con el mismo identificador siempre que las secuencias de tipos de sus parámetros sean diferentes. El compilador de *Java* es capaz de decidir a qué método se está invocando en función de los parámetros que se le pasan. A esta propiedad, que permite usar un mismo nombre y que en cada caso tenga diversas interpretaciones, se le denomina **polimorfismo estático** o **sobrecarga**, y se resuelve de manera estática en la compilación.

La sobrecarga suele utilizarse para dotar de homogeneidad a los interfaces, al permitir que los métodos que hagan algo similar con diferentes parámetros de entrada puedan compartir el mismo identificador.

A diferencia de otros lenguajes como *C++*, en la definición de un método de *Java* no se pueden definir valores por defecto para sus parámetros. Sin embargo, gracias a la sobrecarga, se puede realizar variaciones de un método con diferentes parámetros obteniendo la misma funcionalidad. Por ejemplo, el siguiente fragmento de código define la función `frenar` dentro de la clase `Coche`, utilizando por defecto una reducción de 1 en la velocidad. Además, existe otro método `Frenar` que realiza el frenado utilizando el valor que se le pasa como parámetro.

```
/**
 * Declaración de la clase coche
 * @version 5.0
 */
 class Coche
 {
    private int velocidad;

    /**
    * Acelera el coche
    */
    public void acelerar() {
        velocidad++;
    }

    /**
    * Frena el coche
    */
    public void frenar() {
        frenar(1);
    }

    /**
    * Frena el coche una cierta cantidad
    * @param v cantidad de velocidad a frenar
    */
    public void frenar(int v) {
        velocidad -= v;
    }
 }
```

2.3.3 Creación de objetos

Se ha explicado que para crear un objeto en *Java* se usa la palabra reservada `new` seguida del nombre de la clase y unos paréntesis. En esta operación se está invocando al **constructor** de la clase. Los constructores son métodos especiales que se ejecutan cuando se crea un objeto y que se utilizan para iniciar las propiedades del objeto. De hecho, las propiedades finales solo se pueden modificar en los constructores y en los bloques de inicialización de los que hablaremos más adelante.

Los constructores, como todos los métodos, pueden tener parámetros, aunque no pueden declarar ningún tipo de retorno, y se distinguen porque tienen el mismo nombre que la clase a la que pertenecen. Normalmente, una clase puede tener varios constructores, aunque no puede tener dos constructores que reciban los mismos parámetros (es decir, con el mismo número de parámetros, de los mismos tipos y en el mismo orden).

Si en el código fuente de una clase no se define ningún constructor, *Java*, al compilar, añade un **constructor por defecto** que no tiene parámetros. Este constructor no implementa ningún código y su definición no aparece de manera explícita en el código. Sin embargo hemos visto que se utiliza explícitamente cuando se crean objetos. Cuando en el código de una clase se define uno o más constructores, *Java* no añade el constructor por defecto. Si se desea mantener un constructor sin parámetros se debe definir explícitamente en el código de la clase.

A continuación se muestra un ejemplo de un constructor para la clase Coche:

```
/**
 * Declaración de la clase coche
 * @version 5.1
 */
 class Coche
 {
    private parado = true;
    private final int matricula;
    private int velocidad;

    /** Construye coches asignando valores a algunas de sus propiedades  */
    public  Coche(int v, boolean enMovimiento, int numMatricula) {
        velocidad = v;
        parado = !enMovimiento;
        matricula = numMatricula; //Una propiedad final se inicia en un constructor
    }
    // Si se desea usar el constructor sin parámetros es necesario definirlo explícitamente
    /** Construye coches asignando valor aleatorio a la matrícula
    */
    public  Coche() {
        matricula = rand();     // El atributo velocidad se inicializa con un valor por defecto aleatorio
    }
 }
```

2.3.4 Destrucción de objetos

La destrucción de objetos se realiza de manera automática mediante un mecanismo conocido como la recolección de basura. Para ello la máquina virtual de *Java* revisa de manera periódica los bloques de memoria reservados buscando aquellos que no están siendo referenciados por ninguna variable para liberarlos. La tarea que realiza esta operación se llama **recolector de basura** (***garbage collector***) y se ejecuta en segundo plano intentando aprovechar los tiempos de baja intensidad de proceso.

Podría pensarse que un sistema de liberación de memoria explícito (como el de C++) puede ser mejor que uno basado en recolección automática. Debe tenerse en cuenta que en un sistema de hardware multiproceso, la recolección de basura podría realizarse en un hilo aparte , lo cual haría que no se robase tiempo de proceso al hilo que ejecuta el programa principal.

Tampoco debe creerse que la recolección automática de basura elimina la posibilidad de que se produzcan perdidas de memoria. Es cierto que la memoria nunca se pierde, en el sentido de no liberar memoria que no es apuntada por ningún objeto. Sin embargo, la memoria puede llenarse de objetos que aunque ya no son útiles, aún se mantienen al estar apuntados por otros.

Finalizadores

Los finalizadores son métodos que se ejecutan antes de la liberación del espacio de memoria de un objeto. En ellos se pueden realizar tareas como avisar a otros objetos relacionados de la destrucción de éste. Para añadir un finalizador a una clase basta con añadir un método que se llame `finalize` y cuyo tipo devuelto sea `void`. En general se recomienda no utilizar finalizadores dado que no es posible conocer el momento exacto de su ejecución.

2.3.5 Bloques de inicialización

Java permite definir bloques de inicialización. Estos bloques se definen mediante ámbitos anónimos en el interior de las clases, y se ejecutan de manera previa a cualquier constructor siguiendo el orden en el que estén presentes. Habitualmente se utilizan para implementar código de inicialización común a todos los constructores.

Además, si un bloque está etiquetado como `static` se ejecuta sólo una vez durante la construcción de la clase. Siendo en este caso su principal utilidad la de iniciar variables estáticas de la clase que requieran de algún tratamiento complejo.

Por ejemplo, se puede considerar que un coche siempre está parado cuando se crea. En este caso se podría definir un bloque de inicialización como el siguiente.

```
/**
 * Declaración de la clase coche
 * @version 5.2
 */
 class Coche {
    private int velocidad;
    private final int numRuedas;
    private final int numPuertas;
    private boolean parado;
    public int numUnidadesVendidas;

    /** Constructor que permite construir un coche iniciando ciertos parámetros
     *   @param v velocidad
     *   @param enMovimiento
     *   @param nPuertas Puertas del coche */
```

```
    public  Coche(int v, boolean enMovimiento, int nPuertas) {
        velocidad -= v;
        parado = !enMovimiento;
        numPuertas = nPuertas;
    }

    /** Constructor que permite construir un coche sin pasar parámetros
    */
    public  Coche() {
        // Los atributos se inicializan a los valores por defecto
    }

    // Bloque de inicialización común a todos los constructores
    // Las propiedades finales también se pueden iniciar aquí
    {
        enMarcha = false;
        numRuedas = 4;
    }

    // Bloque de inicialización estático que solo se ejecuta una vez
    static {
        numUnidadesVendidas = 0;
    }
}
```

2.3.6 Los modificadores de control de acceso a los miembros en detalle

Estos modificadores son herramientas que proporciona *Java* para facilitar la encapsulación. Se utilizan al definir cada miembro y especifican la visibilidad de ese miembro desde otras clases. Los modificadores de control de acceso son:

public.- Si la clase `A` tiene un miembro declarado como `public` ese miembro es accesible desde cualquier clase que vea la interfaz de `A`.

private.- Si la clase `A` tiene un miembro declarado como `private` ese miembro sólo es accesible desde los métodos de la clase `A`.

protected.- Si la clase `A` tiene un miembro declarado como `protected` ese miembro es accesible desde los métodos de la clase `A`, por las clases que hereden de `A`, y por las clases definidas en el mismo paquete.

Si no se especifica ningún modificador de control de acceso el miembro declarado es de tipo ***friendly***. Estos miembros son visibles desde cualquier clase que pertenezca al mismo paquete, siendo inaccesibles desde fuera del paquete.

2.3.7 Los modificadores de uso de los miembros en detalle

Estos modificadores se utilizan en la definición de los miembros y permiten especificar características de la implementación de un miembro.

abstract.- Este modificador sólo es aplicable a métodos. Cuando un método se declara `abstract` en una clase `A` ese método no tiene implementación en `A`. Además la clase `A` pasa a ser abstracta. Una clase abstracta es una clase que tiene uno o más métodos abstractos. Cuando una clase tiene todos sus métodos abstractos se dice que es una clase abstracta pura. No se pueden crear objetos de clases abstractas, ya que tienen métodos no definidos. Más adelante se profundizará en la utilidad de las clases abstractas.

En *UML* los métodos abstractos y las clases que los contienen se distinguen porque el texto correspondiente está en cursiva.

static.- Un miembro definido como `static` no pertenece a ninguna de las instancias que se puedan crear de una clase sino a la clase misma. Se dice que los miembros declarados con `static` son miembros de clase, mientras que el resto son miembros de instancia. Así, una propiedad `static` de la clase `A` es una propiedad cuyo valor es compartido por todos los objetos de la clase `A`. Por otro lado, para llamar a un método `static` de la clase `A` no hace falta ningún objeto de la clase `A`. En general no se recomienda el uso de miembros `static` pues son ajenos a la Programación Orientada a Objetos.

Todo programa en *Java* comienza su ejecución en un método `static` denominado `main`. Este método debe encontrase en una clase que tenga el mismo nombre que el fichero que la contiene. Es necesario que dicho método sea estático porque no se crea ningún objeto al ejecutar un programa y sin embargo la máquina virtual de *Java* lo invoca.

En *UML*, las miembros estáticos se distinguen porque el texto correspondiente está subrayado.

final.- Un miembro se declara como `final` cuando se desea impedir que su valor pueda ser cambiado. En general se recomienda usar este modificador para todas las propiedades y métodos de las clases que se definan.

Las propiedades definidas como `final` son constantes a lo largo de la vida de un objeto. Su valor puede definirse en tiempo de compilación, en tiempo de ejecución (en los llamados bloques de inicialización), e incluso pueden definirse de forma tardía en el constructor. Las referencias a objetos declaradas como `final` no pueden cambiarse, aunque los objetos en sí

mismos sí pueden cambiar su estado. Además, pueden utilizarse combinadas con `static` para definir constantes inmutables para todos los objetos.

Por otro lado, los métodos definidos como `final` no pueden cambiar su implementación en clases derivadas. La declaración de un método como final tiene dos objetivos: fijar el diseño y aumentar la eficiencia. Las clases finales permiten ligadura estática de las funciones lo que redunda en mayor velocidad. También permiten fijar razones de diseño al impedir cambios.

native.- Permite utilizar funciones externas a *Java*. Así, los miembros declarados `native` se implementan en otro lenguaje nativo de la máquina en la que se invoca y se asocian a *Java* utilizando bibliotecas de enlace dinámico mediante la *Java Native Interface* (*JNI*).

transient.- El valor de una propiedad definida como `transient` no se desea que se preserve si el objeto tiene la capacidad de persistencia.

synchronized.- Este modificador es aplicable a métodos o ámbitos. Provoca que el código así etiquetado sólo pueda estar siendo ejecutado por un hilo en cada momento para cada objeto.

volatile.- Una propiedad así definida indica al compilador que su uso no debe ser optimizado. Esto se hace para evitar problemas cuando la variable pueda ser utilizada desde varios *threads* de manera simultánea.

strictfp.- Modificador aplicable a métodos que fuerza a *Java* a utilizar una aritmética flotante independiente del procesador para asegurar compatibilidad multiplataforma.

2.3.8 Modificadores en la definición de clases generales

Las clases pueden definirse de diferentes tipos:

public.- Se utiliza el modificador `public` para indicar que la clase es visible desde otros paquetes diferentes al paquete en el que se implementa. Por el contrario una clase no definida con `public` sólo es visible en el paquete en el que se implementa. Sólo puede usarse el modificador `public` por fichero y el nombre de la clase sobre la que se use debe coincidir con el nombre del fichero.

abstract.- Una clase abstracta es aquella que tiene al menos un método abstracto y de la que, por tanto, no pueden crearse instancias. Las clases abstractas deben utilizar el modificador `abstract` para indicar su carácter.

final.- Son aquellas de las que no se puede heredar para crear una nueva clase por estar marcadas con el modificador `final`. En general se recomienda utilizar el modificador `final` para todas las clases.

2.3.9 Clases internas

Una clase interna es una clase que se declara dentro de otra. Para ser más precisos, su declaración se puede realizar dentro del ámbito de la clase o incluso dentro de un método. Su principal utilidad consiste en encapsular la definición de la clase para restringir su uso o su visibilidad.

```
class Coche {
    ...
    class Rueda {
        ...             //La clase rueda es interna a la clase Coche
    }
}
```

En general las clases internas tienen varias particularidades:

- No se permite declarar miembros estáticos en ellas.
- Para crear un objeto de una clase interna es necesario que se haga desde una instancia de la clase contenedora.
- Desde una clase interna es posible acceder a las propiedades de la clase contenedora como si fuesen propias.
- Para declarar una referencia del tipo de una clase interna fuera de la clase contenedora debe usarse la sintaxis `ClaseContendora.ClaseInterna`.

En la definición de clases internas es posible añadir los modificadores `public`, `private` o `protected`. Si no se añade ningún modificador la clase será visible en el paquete, si se añade `public` será visible en general, si se añade `private` sólo será visible dentro de la clase en la que está definida y si se añade `protected` sólo será visible dentro de la clase en la que está definida y en las clases que de ella deriven.

También es posible declarar una clase interna con el modificador `static`. Las clases internas así definidas se diferencian del resto de clases internas en 3 particularidades:

- En estas clases internas se pueden definir miembros estáticos.
- Desaparece la posibilidad de acceder a las propiedades de la clase contenedora.
- También desaparece la restricción que impide crear objetos de la clase interna sin un objeto de la clase contenedora.

Por último, no se puede cerrar este apartado sobre clases internas sin hablar de las **clases anónimas**. Son clases que no tienen identificador. Se utilizan cuando se hereda de otra clase, no se desea añadir nada a su interfaz y solo se quiere crear un único objeto.

2.3.10 Ejemplo del parking

Aquí se presenta como ejemplo la codificación de una clase que implementa una estructura de datos de tipo pila para almacenar objetos de la clase `Coche`. Las pilas son unas estructuras de datos clásicas que se caracterizan por almacenar los elementos de manera que el último en ser insertado es el primero en ser recuperado.

Una forma de implementar las pilas consiste en utilizar una sucesión de objetos contenedores enlazados mediante referencias. En este caso se ha creado la clase `Plaza` como contenedor de objetos `Coche`. Como se puede apreciar en el diagrama estático adjunto, la clase `Parking` tiene una referencia a un objeto de la clase `Plaza`, el cual a su vez tiene una referencia a un siguiente objeto `Plaza`, y así sucesivamente. Es importante notar que la siguiente a la última plaza tomará valor `null`. Obsérvese también, que la clase `Plaza` se ha definido interna a la clase `Parking`.

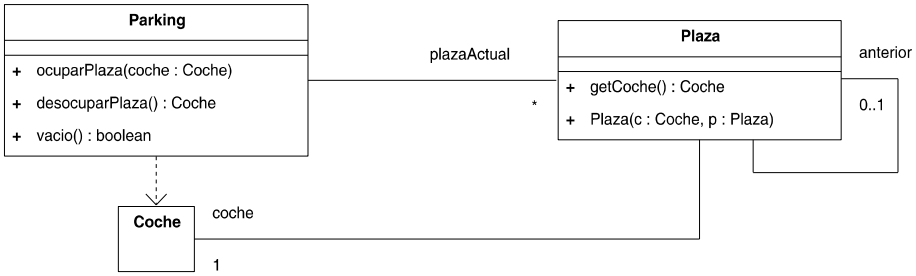


Figura 18.- Diagrama estático de la estructura del `Parking`.

```
/**
 * Definimos la clase Parking. Este es un Parking ordenado, es decir, los
 * coches no deciden que plaza ocupan, el parking les asigna la libre más próxima.
 * Además, tiene plazas infinitas (en una estructura de tipo pila)
 * La hacemos final porque no queremos que nadie pueda heredar de esta clase
 * @version 1.0
 */
public final class Parking {

  //Como la clase Plaza solo interesa a Parking, se define interna, privada y final
  private final class Plaza {

      private final Plaza anterior;
      private final Coche coche;

      private Plaza(Coche nuevoCoche, Plaza plazaAnterior) {
          anterior = plazaAnterior;
          coche = nuevoCoche;
      }

      public getCoche() {
           return coche;
      }
  }

  //La plaza actual, inicialmente a null, acepta coches en el parking.
  //No hacemos la propiedad final porque cambia.
  private Plaza plazaActual;

  /**
   * Inserta un coche al parking
   */
  public void ocuparPlaza(Coche coche) {
      //Incrementamos las plazas ocupadas
      plazaActual = new Plaza(coche, plazaActual);
  }

  /**
   * Comprueba si el parking está vacio
   */
  public boolean vacio() {
     return (plazaActual == null);
  }
```

```
    /**
     * Saca un coche del parking
     */
    public Coche desocuparPlaza() {
        //Al desocupar la última plaza, la nueva última plaza ocupada es la anterior
        Coche coche = plazaActual.getCoche();
        plazaActual = plazaActual.anterior;
        return coche;
    }
}
```

2.3.11 Definición de *arrays* de objetos y de tipos primitivos

Java proporciona una clase ***array*** como contenedor básico de objetos y tipos primitivos. Para la creación de objetos *array* en *Java* se ha sobrecargado el operador corchetes. Así, para la declaración de una referencia a un objeto *array* se utiliza el tipo de objetos o tipo primitivo que contendrá el *array* seguido de una pareja de corchetes vacía. Como siempre, si una referencia no se inicializa su valor es `null`. Para crear un *array*, se utiliza la palabra reservada `new` seguida de una pareja de corchetes con las dimensiones del *array*[6]. Hay que resaltar que una vez dimensionado un *array* no es posible redimensionarlo.

```
<tipo> '[]' <identificador> = new <tipo > [<dimensión>];
<tipo> '[][]' <identificador > = new <tipo > [<dimensión1>] [<dimensión2>];
<tipo> ['[]']* <identificador > = new <tipo > [<dimensión1>] ...[<dimensiónN>];
```

El siguiente ejemplo muestra un *array* de enteros y un *array* de objetos de la clase `Coche`.

```
int [] vector = new int[25];
Coches [] aparcamiento = new Coches [30];
```

Java también permite declarar implícitamente la dimensión de un *array* inicializándolo con los elementos que contiene y sin especificar su dimensión.

```
<tipo> '[]' <identificador> = {[objetos|valores primitivos|cadenas de caracteres]*};
```

Por ejemplo una declaración implícita de la dimensión con inicialización sería:

```
int [] fib = {1,1,2,3,5,8,13};
```

El acceso a un elemento de un *array* se realiza utilizando la variable y entre corchetes el índice del elemento al que se desea acceder. Los índices comienzan por cero y alcanzan como máximo un valor igual a la dimensión del *array* menos uno.

Este primer ejemplo crea un *array* de enteros y lo rellena con el valor 3.

```
int [] vector = new int[25];

for (int x = 0; x < 25; x++)
   vector[x] = 3;
```

El siguiente ejemplo crea una tabla bidimensional de objetos de la clase `Coche` y una referencia a la misma. Luego inicializa cada referencia de la tabla con un objeto `Coche` nuevo.

```
Coche [][] tablaCoches = new Coche[25][30];

for (int x = 0; x < 25; x++)
for (int y = 0; y < 30; y++)
   tablaCoches[x][y] = new Coche();
```

Java también permite definir un *array* compuesto de otros *arrays* de tamaños variables. Esta posibilidad se potencia con el uso de la propiedad `length` que tiene la clase *array* y que devuelve la dimensión del mismo. Por ejemplo, un *array* bidimensional como el presentado en el esquema de la Figura 19 correspondería al siguiente código.

[6] Por herencia de *C*, *Java* también permite definir primero el identificador y luego los corchetes, aunque es preferible la primera opción al quedar más claro cual es el tipo de la variable.

```
int [][] vectores = new int[2][];
vectores[0] = new int[5];
vectores[1] = new int[8];

for (int x = 0; x < tabla.length; x++)
for (int y = 0; y < tabla[x].length; y++)
      vectores[x][y] = -1;
```

Por último se debe señalar que para los *arrays* compuestos de caracteres se permite una sintaxis especial.

```
char [] cadena = "Esto es un array\nde chars";//Array de chars con retorno de carro
```

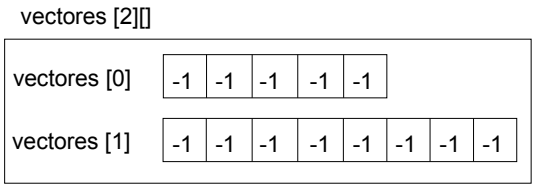


Figura 19.- Representación esquemática de un *array* bidimensional de enteros.

2.4. Los paquetes

Java estructura las bibliotecas de *software* mediante paquetes. Los paquetes son agrupaciones lógicas de código y constituyen el mecanismo que proporciona *Java* como soporte a la modularidad (ver capítulo 1).

Los paquetes se organizan entre ellos en forma de árbol y esta organización repercute en su nombre. Por ejemplo, un paquete podría ser `es.urjc.software` y otro paquete contenido en éste podría ser `es.urjc.software.utilidades`.

Las organización en árbol es un mecanismo que proporciona *Java* para facilitar la jerarquía (ver capítulo 1). Cada vez que se crea un paquete se está especificando implícitamente una estructura de directorios. El paquete debe residir en el directorio indicado por su nombre, y debe ser un directorio localizable a partir de alguno de los directorios declarados en la variable de entorno `CLASSPATH` (o en el parámetro `classpath` en la invocación a la Máquina Virtual de *Java*).

Por otro lado, la pertenencia de un fichero a un paquete se debe declarar en la primera línea de código del fichero mediante la palabra reservada `package` seguida del nombre del paquete al que pertenece.

Es posible acceder a un elemento contenido en un paquete si se antepone el nombre completo del paquete al nombre del elemento (lo que se conoce como **nombres completamente calificados**). Así, para crear un objeto de la clase `Coche` contenida en el paquete `Utilitarios` del paquete `Vehiculos` debemos escribir:

```
vehiculos.utilitarios.Coche  miCoche = new vehiculos.utilitarios.Coche();
```

También se puede utilizar la palabra reservada `import` al principio de un fichero para evitar usar nombres completamente calificados.

```
import vehiculos.utilitarios.Coche;

...

Coche miCoche = new Coche();
```

Incluso es posible utilizar el asterisco para especificar que se desea importar todas las clases contenidas dentro de un determinado paquete.

```
import vehiculos.utilitarios.*;

...

Coche miCoche = new Coche();
```

También es posible utilizar la combinación `import static` para invocar los métodos estáticos de una clase sin necesidad de utilizar nombres completamente calificados.

Resta decir que una clase siempre pertenece a un paquete. Por eso, cuando se define una clase, si no se especifica ningún paquete se considera que están en el **paquete por defecto**, que es un paquete asociado al directorio de ejecución que se crea automáticamente. Esta práctica está desaconsejada porque va en contra de cualquier principio de organización. Si la variable `CLASSPATH` está definida será necesario que contenga el directorio donde se encuentran los ficheros del paquete por defecto. Si no está definida no es necesario.

2.5. Lecturas recomendadas

"El lenguaje de programación *Java*", 4ª Edición, Ken Arnold, James Gosling, David Holmes, Addison Wesley, 2001.

"Introducción a la Programación Orientada a Objetos con *Java*", C. Thomas Wu, Mc Graw Hill, 2001.

2.6. Ejercicios

Ejercicio 1

¿Cuál es la diferencia entre una propiedad de clase y una propiedad de instancia?

Ejercicio 2

¿Por qué no existe en *Java* una instrucción para la eliminación de la memoria reservada con `new` (como `delete` en *C++*, o `dispose` en *Pascal*)?

Ejercicio 3

El siguiente fragmento de código, que corresponde a una clase `A`, contiene varios errores. Detallar en que consisten los errores, indicando cada vez la línea en la que se producen.

```
1  public B metodo1(B c)
2  {
3     B aux = new B();
4     A aux2;
4     aux = c;
5     aux.Añadir(25);
6     if (aux = c)
7        aux2.Value(25);
8     return aux;
9     aux2 = null;
10 }
```

Capítulo 3 Herencia y genericidad en Java

En este capítulo se introducen los aspectos más relevantes de la forma en que *Java* da soporte a la herencia y a la genericidad.

3.1. La herencia de clases

Java permite la herencia simple de clases. Básicamente la herencia permite que una clase copie de otra su interfaz y su comportamiento y que añada nuevo comportamiento en forma de código. La herencia facilita la reutilización del código sin afectar demasiado a la deseable propiedad de encapsulación.

Más concretamente, la herencia de clases posibilita que los miembros públicos y protegidos de una clase `A` sean públicos y protegidos, respectivamente, en cualquier clase `B` descendiente de `A`. Por ello, los miembros públicos y protegidos de una clase se dice que son accesibles desde las clases que derivan de ella. Por otro lado, los miembros privados de una clase `A` no son accesibles en los descendientes de `A`.

Para añadir nuevas funcionalidades, sobre una clase `B` que herede de una clase `A` se pueden implementar tantos métodos nuevos como se desee.

En el punto 2.3 se dijo que la definición de una clase se correspondía con:

```
<clase> ::=
  [Modificador de clase] class <identificador> [parámetros] [herencia] [excepciones]
{
  [<método>|<propiedad>|<clase>|<bloque inicialización>]*
}
```

El elemento herencia debe sustituirse por:

```
 <herencia> ::= extends <super clase>
```

El siguiente ejemplo ilustra la herencia.

```
/**
 * Declaración de la clase coche
 * @version 6.0
 */
 class Coche {
    public int matricula;
    private int ruedas;

    private void revisar()     {
        //El método privado no será heredado
    }
    protected void frenar()    {
        //El método protegido sí se hereda
    }
    void acelerar() {
        //Un método friend se hereda si está en el mismo paquete
    }
    public void abrirPuerta() {
        //El método público también se hereda
    }
 }
```

```
//La clase Deportivo deriva de Coche
class Deportivo extends Coche {
    private int inyectores = 12;
    public void turbo() {
        /*
        VIOLACIONES DE LA VISIBILIDAD
        ruedas++;       // Error, las propiedades privadas no son visibles
        revisar();      // Error, método no visible fuera de Coche
        */

        frenar();               //Correcto, Frenar es visible en derivados de Coche
        abrirPuerta();          //Correcto, Revisar es visible siempre
        inyectores++;           //Correcto, inyectores es visible en Deportivo
        acelerar();             //Correcto, Acelerar es visible en el paquete
    }
}

class Prueba {
    public static void main (String [] arg)     {
        Deportivo d = new Deportivo();

        /*
        VIOLACIONES DE LA VISIBILIDAD
        d.inyectores++;         //Error, propiedad no visible fuera de Deportivo
        d.revisar();            //Error, método no visible fuera de Coche
        d.frenar();             //Error, método no visible fuera de Deportivo
        */

        d.matricula = 5;        //Correcto, las propiedades públicas son accesibles
        d.abrirPuerta();        //Correcto, el método es visible dentro del paquete
        d.turbo();              //Correcto, los métodos públicos son accesibles
        d.revisar();            //Correcto, los métodos públicos se heredan
    }
}
```

La herencia posibilita que un objeto cumpla varios tipos. En efecto, como cada clase define un tipo, un objeto de una clase `A` cumple el tipo definido por `A` y también cumple los tipos definidos por todas las clases de la jerarquía de la que deriva `A`.

Es importante notar que en *Java* cada clase no puede heredar de más de una clase simultáneamente. Es decir, no permite herencia múltiple, sólo herencia simple. Las jerarquías de herencia de clases de *Java* tienen forma de árbol. De hecho, en *Java* existe una clase llamada `Object` de la que deriva toda clase que no declare derivar de otra. Por tanto, en *Java* hay un único árbol de herencia, y `Object` es superclase de todas las clases de *Java*.

3.1.1 Herencia en clases internas

Las clases internas también pueden heredar de otras clases. En particular, las clases internas estáticas pueden utilizar la herencia, consiguiendo efectivamente que una clase herede de manera aislada diferentes partes de su interfaz de jerarquías diferentes.

3.1.2 Redefinición de miembros

En una clase `B` que deriva de una clase `A` es posible redefinir los miembros accesibles de `A` implementándolos de nuevo. Con esto se consigue que `B` redefina el comportamiento definido por su superclase `A`.

Cuando un método `M` se redefine en una clase `B` que deriva de `A`, cualquier llamada al método `M` sobre un objeto de `B` produce una llamada a la implementación de `B` y no a la de `A`, que queda enmascarada.

Es posible acceder a la implementación de la superclase dentro de un miembro redefinido mediante el uso de la propiedad privada `super` que no es otra cosa que una referencia a la interfaz de la superclase.

A partir de la versión 5 de *Java* es posible que el tipo devuelto por el método redefinido pueda ser un tipo derivado del tipo devuelto por el método de la superclase.

3.1.3 La herencia y los constructores

Los constructores accesibles de una clase `A` son visibles desde cualquier clase `B` derivada de `A`, pero no se puede construir objetos de la clase `B` si no se redefine los constructores en `B`. Es decir los constructores siempre se tienen que implementar, aunque desde un constructor de una clase `B` se puede llamar al constructor de la superclase utilizando la propiedad privada `super`.

Si al definir un constructor en una clase derivada no se llama al constructor de la clase padre *Java* añade una llamada al constructor por defecto de la superclase. En el caso de que la superclase no tenga constructor por defecto *Java* da un error de compilación porque al crear un objeto no sabrá como inicializar los parámetros de la superclase.

Supóngase una clase `A` con dos constructores.

```
class A   {
   public A() {
      //Implementación
   }

   public A(int x) {
      //Implementación
   }
}
```

El siguiente fragmento crea una clase `B` que deriva de la clase `A` anterior. Esta clase tiene dos constructores. El primero no invoca al constructor de la superclase, por lo que se invoca automáticamente al constructor por defecto. El segundo invoca a uno en especial de los constructores de la superclase.

```
class B extends A {
   public B() {
      //No se especifica constructor por lo que automáticamente se
      //llama al constructor por defecto de la superclase
      //resto de la implementación
   }

   public B(int x) {
      super(x);      //Llamada al constructor con parámetros de la superclase
      //resto de la implementación
   }
}
```

Si en la clase `A` el método `M` es público no es posible ocultarlo en ninguna clase `B` derivada de `A` redefiniendo `M` como privado. El intento de ocultar un método que es público en una superclase da un error de compilación pues implica que se intenta violar el contrato de la clase padre.

Ejemplo de herencia sobre la clase `Parking`

El siguiente ejemplo hereda del `Parking` del ejemplo 2.3.10 para implementar un `Parking` que solo admite coches con matrícula par. Para ello redefine el método `ocuparPlaza`, de manera que solo invoca a la superclase cuando la matrícula es par.

```
class ParkingPares extends Parking {
   public void ocuparPlaza(Coche c) {
      if (c.matricula % 2 == 0)
         super.ocuparPlaza (c);
   }
}
```

3.2. Polimorfismo dinámico

En *Java* un objeto de un tipo `T` puede estar referenciado por una referencia del tipo `T` o por una referencia de cualquier tipo del que `T` derive.

Así, *Java* permite asignar el contenido de una referencia de una clase `B` a una referencia de una superclase de `B`. Por ejemplo si la clase `B` deriva de la clase `A` se puede hacer:

```
B b = new B();
A a;
a = b;
```

Este mecanismo permite que una misma referencia pueda apuntar en la misma ejecución a objetos de clases diversas. Cuando esto se une a la redefinición de métodos se da la circunstancia de que en compilación no puede decidirse a qué implementación se invoca cuando se llama a un método. Esta decisión se tomará en ejecución dependiendo del objeto sobre el que se invoque. Como ya se explicó en el capítulo 1, a esta propiedad, que permite usar un mismo nombre y que en cada caso provoca diferentes ejecuciones, se la denomina polimorfismo dinámico, ya que se resuelve en ejecución.

El siguiente ejemplo ilustra este comportamiento. Supóngase que la clase `C` y la `B` derivan de la clase `A`. Supóngase además, que la clase `A` tiene un método `M` que deja para sea implementado por la clase `B` y la `C`. Cuando en el fragmento de código adjunto se invoca al método `M` es imposible predecir a priori si se ejecuta la implementación de la clase `B` o de la clase `C`.

```
A a;

if (Math.random() > 0.5)
   a = new B();
else
   a = new C();
a.M();
```

Ejemplo del juego de coches

Supongamos que deseamos construir un juego de simulación de tipo persecución de coches. Así tendríamos deportivos, ambulancias, camiones... Todas estas clases podrían derivar de una clase `Automóvil`, de manera que se aprovechase la definición común de parte de la implementación. Cada clase añadiría los detalles relativos a esa clase particular. Así la clase ambulancia añadiría el método `encenderSirena()`, o la clase camión podría tener la propiedad booleana `estaCargado`.

Además, si a la clase `Vehículo` añadimos el método abstracto `pintar()` cada clase derivada podrá implementarlo de acuerdo a su carácter. Así la clase `Ambulancia` pintará una furgoneta blanca con una cruz roja, mientras que la clase `Deportivo` podría pintar un *Ferrari* rojo.

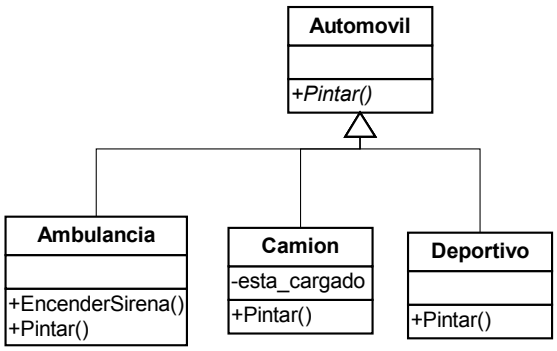


Figura 20.- Jerarquía para el juego de coches.

Queda claro en este ejemplo que cada una de estas clases (`Deportivo`, `Ambulancia`, `Camión`...) son de la clase `Automóvil`, y por lo tanto una referencia de tipo `Automóvil` podría referenciar tanto a un `Deportivo` como a un `Camión`. Es más, desde una referencia del tipo `Automóvil` podríamos invocar al método `pintar()`, aunque a priori no podríamos saber el resultado, pues la referencia puede apuntar a uno u otro de los diferentes tipos de vehículos.

Este comportamiento es altamente útil al realizar programas de una complejidad media o elevada, ya que permite al desarrollado trabajar con abstracciones de un nivel superior, sin preocuparse de los detalles de las abstracciones de nivel inferior. Así, por ejemplo, podríamos tener un objeto encargado de refrescar la pantalla recorriéndose una estructura genérica que almacena objetos de la clase Vehículo. Tal objeto no es necesario que conozca la interfaz particular de los diferentes tipos de vehículos, sólo hace falta que invoque el método `Pintar` de cada `Automóvil` que contenga la estructura.

Una implementación para las clases descritas podría ser:

```
class Automóvil {
   public void pintar();
}
class Ambulacia extends Automóvil {
   public void pintar() {
      System.out.println("Soy una ambulacia");
   }
   public void encenderSirena() {
      System.out.println("IIIIIIIIIAAAAAAAAAA");
   }
}
class Camion extends Automóvil {
   private boolean estaCargado = false;
   public void pintar() {
      if (estaCargado)
         System.out.println("Soy un camión cargado");
      else
         System.out.println("Soy un camión");
   }
}
class Deportivo extends Automóvil {
   public void pintar() {
      System.out.println("Soy un Deportivo");
   }
}
```

3.2.1 El *casting* de referencias a objetos

En *Java* no se permite asignar directamente el contenido de una referencia de un tipo `T` a una referencia de un tipo derivado de `T`.

Sobre el ejemplo anterior se ve claramente que un objeto de la clase `Ambulancia` es necesariamente un objeto de la clase `Automóvil`. Nótese que el reciproco no es cierto: un objeto de la clase `Automóvil` no tiene porque ser precisamente de la clase `Ambulancia` (pues podría ser otra clase derivada de `Automóvil`). Es para prevenir este tipo de errores por lo que no se permite asignar directamente el contenido de una referencia de un tipo `T` a una referencia de un tipo derivado de `T`.

Sin embargo, aunque la asignación mencionada no se puede realizar directamente, sí se puede hacer cuando previamente se realiza una operación de *casting*.

El siguiente ejemplo ilustra el uso del *casting*:

```
Ambulancia a = new Ambulancia();
Automóvil b = a;
Ambulancia c = (Ambulancia) b; //Aquí se hace el casting
```

El *casting* indica al compilador un tipo al que deberá pertenecer el objeto referenciado por `b` cuando se ejecute el programa. Obsérvese que debido al polimorfismo el compilador no puede comprobar el tipo en compilación. Por ello, si durante su ejecución el objeto no pertenece al tipo especificado en el *casting* se producirá un error en forma de excepción.

Debido a la posibilidad de que en ejecución se generen este tipo de errores, una buena norma sería no utilizar el *casting* nunca. Desgraciadamente su uso es imprescindible en algunas ocasiones. Por ejemplo, al recuperar un objeto de un contenedor genérico y querer poder acceder a la interfaz específica del objeto recuperado. En estos casos debe minimizarse su utilización, encapsulando el *casting* dentro de unos pocos métodos para que sólo se utilice en partes muy controladas del código.

En *Java* puede usarse la palabra reservada `instanceof` para comprobar la pertenencia de un objeto a una clase y con ello asegurar un correcto funcionamiento del *casting*.

```
if (x instanceof Coche) {
   Coche c = (Coche) x;
}
```

Sin embargo este tipo de comprobaciones debe evitarse en lo posible, pues hace que se creen dependencias fuertes entre clases que lógicamente no deberían depender entre si. Así, el uso de `instanceof` es en general contrario a la modularidad y normalmente es innecesario si se utiliza adecuadamente el polimorfismo.

Ejemplo del vector de *objects*

Se desea construir una clase `VectorDinamico` que permita almacenar objetos de clase `Object` o de cualquier clase que herede de ésta (y por tanto de cualquier clase de objeto).

```
public class VectorDinamico {

   private int dimension = 0;

   private Object array[] = null;

   VectorDinamico(int dimension) {
      redimensionar(dimension);
   }

   public int dimension()     {
      return dimension;
   }

   public void poner(int pos, Object o) {
      if (pos >= dimension)
         redimensionar(pos+1000);
      array[pos] = o;
   }

   public void redimensionar(int dimension) {
      if (this.dimension >= dimension)
         return;

      Object nuevo_array[] = new Object[dimension];

      for (int cont = 0; cont < this.dimension; cont++)
         nuevo_array[cont] = array[cont];

      array = nuevo_array;
```

```
        dimension = this.dimension;
    }

    public Object obtener(int pos) throws Exception {
        if (pos < dimension)
            return array[pos];
        else
            throw new ErrorVector("VectorDinamico.obtener(): Está vacío");
    }
}

class ErrorVector extends Excepction {
    public String error;

    ErrorVector(string error) {
        this.error = error;
    }
}
```

El clase `VectorDinamico` permitirá almacenar cualquier tipo de objeto, pero al recuperarlo se deberá hacer *casting* a una clase concreta para poder utilizar el objeto. Esto puede hacer azaroso el uso de una estructura genérica, ya que al recuperar un objeto de la estructura no se sabe con seguridad de que tipo es el objeto. Una buena solución es controlar el acceso a tal estructura de manera que sólo sea posible insertar en ella objetos de un tipo. Esto se puede hacer creando una clase que contenga la estructura `VectorDinamico` como propiedad privada, y que tenga métodos públicos para insertar y obtener elementos del tipo concreto que se desee en la estructura.

Ejemplo del aparcamiento de coches

En el siguiente ejemplo utiliza la clase `VectorDinamico` para crear una nueva clase `Aparcamiento` que permita la inserción y obtención de objetos únicamente de la clase `Coche`.

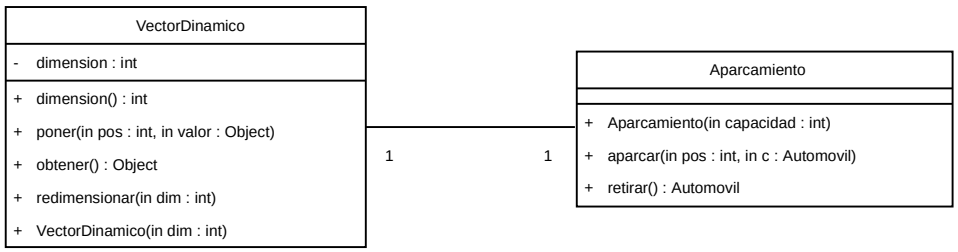


Figura 21.- Relación de asociación entre el vector dinámico y el aparcamiento.

```
class Aparcamiento {
  private VectorDinamico datos;

  public Aparcamiento(int dimension) {
    datos = new VectorDinamico(dimension);
  }

  public void aparcar(int pos, Automóvil c) {
    datos.poner(pos, c);
  }

  public Coche retirar(int pos) {
    return (Coche) datos.obtener(pos);
  }
}
```

Obsérvese que no es posible usar herencia para que `Aparcamiento` herede de `VectorDinamico`, ya que la clase `Aparcamiento` heredaría los métodos que permiten insertar cualquier `Object` y no se habría conseguido nada.

3.3. Las *interfaces*

Se ha dicho que cuando se define una clase de objetos se define la interfaz y la implementación de los objetos que cumplen dicha clase. Hay que añadir que *Java* también permite definir una interfaz sin tener que definir su implementación. La declaración de una interfaz se realiza mediante la siguiente estructura.

```
interface <id_interfaz> [extends <id_interfaz> [<id_interfaz>]*] ]
{
    [propiedades estáticas finales | declaración de métodos | interfaz interna]*
}
```

Las interfaces se pueden ver como clases abstractas puras. Es decir, clases que sólo declaran métodos y propiedades estáticas finales, y que no definen el comportamiento de ningún método. Así, cualquier propiedad definida en una interfaz es estática y final, mientras que cualquier método es abstracto y público. La definición de interfaces constituye el segundo mecanismo que proporciona *Java* para definir tipos.

Java permite que una clase declare que implementa una o más interfaces simultáneamente, lo cual equivale a la herencia múltiple de clases abstractas. Cuando una clase declara que implementa una o varias interfaces indica con ello que se responsabiliza de definir el comportamiento de cada uno de los métodos que declaran esas interfaces. La declaración de que una clase implementa una o varias interfaces sigue la siguiente estructura.

```
class <id_clase> [extends <id_superclase>][implements <id_interfaz>[,<id_interfaz>]*]
{
  [métodos|propiedades]
}
```

Las interfaces potencian el comportamiento polimórfico de *Java*. Esto se debe a la posibilidad de definir referencias a objetos que implementen cierta interfaz, y a la posibilidad ya descrita de que una referencia a un objeto que cumple un tipo `T` pueda corresponder a cualquier tipo derivado de `T`.

Como peculiaridad se comentará que es posible definir interfaces internas, definiéndolas dentro de clases o de otras interfaces. En este caso, para referenciarlas se usará la misma política que para referenciar clases internas. Además, las interfaces internas admiten el uso del modificador `private`, para restringir su visibilidad.

Ejemplo del conjunto ordenado

Un conjunto ordenado es una estructura que permite almacenar sin repeticiones elementos entre los que existe un orden lo que permite que su recuperación sea más eficiente que en una búsqueda secuencial. A este efecto, la estructura podría consistir en un árbol binario en el que se sigue la regla de que los elementos menores están a la izquierda y los mayores a la derecha.

Por otro lado, para establecer un orden entre los elementos de este árbol podría definirse la interfaz `ElementoOrdenable`. Esta interfaz tendría un método que permitiría comparar "elementos ordenables". Así, cualquier clase que implemente la interfaz `ElementoOrdenable` podría ordenar sus elementos respecto a otros elementos que también cumplan dicha interfaz. Definir el criterio de ordenación como una interfaz tiene la ventaja de que permite que se pueda aplicar a cualquier clase ya existente sin más que heredar y declarar que la nueva clase implementa la interfaz `ElementoOrdenable`.

```
interface ElementoOrdenable  {
  public boolean mayorQue(ElementoOrdenable e);
}

class ArbolBinario {
  private ElementoOrdenable eRaiz;
  private ArbolBinario      arbolDerecha, arbolIzquierda;

  public ArbolBinario izquierda()       {
    return arbolIzquierda;
  }

  public ArbolBinario derecha() {
    return arbolDerecha;
  }
  public ElementoOrdenable obtenerElementoRaiz() {
    return e_raiz;
  }
  public void insertar (ElementoOrdenable e) {
    if (eRaiz == null)
       eRaiz = e;
    else {
       if (e.mayorQue(e_raiz)) {
          if (arbolDerecha == null)
             arbolDerecha = new ArbolBinario();
          arbolDerecha.insertar(e);
       }
       else {
          if (arbolIzquierda == null)
             arbolIzquierda = new ArbolBinario();
          arbolIzquierda.insertar(e);
       }
    }
  }
}
```

Por último, se prueba la clase creada mediante el siguiente código de ejemplo.

```
class EnteroOrdenable implements ElementoOrdenable {
  public int elemento;
  public EnteroOrdenable(int e) {
    elemento = e;
  }
  public boolean mayorQue(ElementoOrdenable e) {
    EnteroOrdenable aux = (EnteroOrdenable) e;
    return (elemento > aux.elemento);
  }
}

class EjemploUso {
 public void main(String args[]) {
    EnteroOrdenable a = new EnteroOrdenable(3);
    EnteroOrdenable b = new EnteroOrdenable(5);
    EnteroOrdenable c = new EnteroOrdenable(11);
    EnteroOrdenable d = new EnteroOrdenable(7);
    EnteroOrdenable e = new EnteroOrdenable(13);

    ArbolBinario arbol = new ArbolBinario();
    arbol.insertar(a);
    arbol.insertar(b);
    arbol.insertar(c);
    arbol.insertar(d);
    arbol.insertar(e);
 }
}
```

3.4. La genericidad

La **genericidad** puede definirse como una capacidad que permite a un lenguaje declarar tipos mediante parámetros variables. La genericidad es un mecanismo que suele utilizarse para construir contenedores conscientes del tipo, de manera que se eviten los problemas descritos al crear inseguros contenedores de `Objets`. La genericidad es un mecanismo más débil que el polimorfismo, y ello se constata en el hecho de que *Java* crea la genericidad sobre el polimorfismo.

3.4.1 Clases e interfaces genéricas

Para dar soporte a la genericidad, en la versión 1.5 *Java* introduce el concepto de **genérico** (similar al de *template* presente en *C++* desde su estandarización *ANSI* en 1998). Los genéricos permiten definir parámetros que se corresponden con un tipo en la definición de una clase o una interfaz. Estos genéricos dan lugar al concepto de clase (o interfaz) genérica. Las clases (o interfaces) genéricas también suelen denominarse **clases parametrizadas**.

En el punto 2.3 se definió la implementación de la siguiente manera:

```
<clase> ::= [Modif. de clase] class <identif.> [parámetros] [herencia] [excepciones]
            '{'
               [<método>|<propiedad>|<clase>|<bloque inicialización>]*
            '}'
```

Ahora podemos explicar que el opcional "parámetros" se utiliza para la genericidad y toma la siguiente forma:

```
parametros ::= ['<'identificador [extends tipo][implements <tipo>[&<tipo>]* '>']
               [,'<'identificador [extends tipo][implements <tipo>[&<tipo>]*]]*'>']
```

Obsérvese que a continuación del genérico se pueden especificar restricciones para limitar las clases que se pueden pasar como parámetro en la construcción de objetos. Es más, el tipo que limita al genérico puede ser otro genérico.

Dentro de la definición de la clase se puede utilizar el identificador del tipo para definir otras variables o valores devueltos.

Por otro lado, para crear objetos de una clase parametrizada se debe usar la siguiente sintaxis:

```
 <objeto parametrizado> ::= <clase> '<'param'>' <variable> =
                              new <clase> '<'param'>' ([parámetros])
```

La principal utilidad de las clases parametrizadas es la de permitir la creación de contenedores genéricos que acepten elementos independientes de su tipo. Hasta la versión 1.4 de *Java*, la genericidad se conseguía construyendo contenedores que almacenaban objetos de la clase `Object` y el *casting* posterior a la clase correspondiente. Un ejemplo lo encontramos en la clase `VectorDinamico` del punto 3.2.1. El problema que plantea esta forma de construir contenedores genéricos es que

no se impone ninguna limitación a los objetos al insertarlos en el contenedor. Por ello podría darse la situación de tener un contenedor lleno y no saber de qué clase son los objetos contenidos.

Ejemplo de uso de genéricos

El siguiente ejemplo muestra cómo utilizar los genéricos al crear una clase `Concesionario` genérica, que luego será capaz de generar objetos `Concesionario` de objetos `Deportivo` y objetos `Concesionario` de objetos `Utilitario`.

```
/**
 * Clase Concesionario genérica
 */
class Concesionario<T extends Coche> {
private T[] coches = null;
    /**
     * Constructor del concesionario
     * @param coches array con los coches que contiene el concesionario
     */
    public Concesionario(final T[] coches) {
        this.coches = coches;
    }
    /**
     * Obtiene todos los coches del concesionario
     * @return array con todos los coches del concesionario
    */
    T[] obtenerCoches() {
        return coches;
    }
}
```

A continuación veamos los distintos tipos de coches que se pueden vender (`Deportivo` y `Utilitario`, ambos implementando la interfaz `Coche`):

```
/**
* Interfaz Coche
*/
interface Coche {
    /**
    * Devuelve el tipo de cohe
    * @return tipo de coche
    */
    String tipo();
}

/**
* Clase Deportivo
*/
public class Deportivo implements Coche {
    /**
    * Tipo de coche
    */
    private String tipo;
    /**
    * Constructor de un deportivo
    * @param marca marca del coche
    */
    public Deportivo(final String marca) {
        this.tipo = "Deportivo: " + marca;
    }
    /**
    * Devuelve el tipo de coche
    * @return tipo de coche
    */
    public String tipo() {
        return tipo;
    }
}

/**
* Clase Utilitario
*/
public class Utilitario implements Coche {

    /**
    * Tipo de coche
    */
    private String tipo;

    /**
    * Constructor de un deportivo
    * @param marca marca del coche
    */
    public Utilitario(final String marca) {
        this.tipo = "Utilitario: " + marca;
    }
```

```
    /**
    * Devuelve el tipo de coche
    * @return tipo de coche
    */
    public String tipo() {
        return tipo;
    }
}
```

Finalmente mostramos el uso de la clase `Concesionario` desde un método `main`.

```
public class ConcesionarioMain {
    public static void main(String[] args) {
        //Creamos los coches deportivos
        final Deportivo[] deportivos = {new Deportivo("Ferrari"),
                                        new Deportivo("Porsche")};
        //Creamos el concesionario de coches deportivos
        final Concesionario<Deportivo> concesionarioDeportivo = new
            Concesionario<Deportivo>(deportivos);
        for (Deportivo deportivo : concesionarioDeportivo.obtenerCoches()) {
            System.out.println(deportivo.tipo());
        }
        //Creamos los coches utilitarios
        final Utilitario[] utilitarios = {new Utilitario("Seat"),
                                          new Utilitario("Renault")};
        //Creamos el concesionario de coches utilitarios
        final Concesionario<Utilitario> concesionarioUtilitario =
                        new Concesionario<Utilitario>(utilitarios);
        for (Utilitario utilitario : concesionarioUtilitario.obtenerCoches()) {
            System.out.println(utilitario.tipo());
        }
    }
}
```

La salida para este `main` es:

```
Deportivo: Ferrari
Deportivo: Porsche
Utilitario: Seat
Utilitario: Renault
```

Al crear una clase parametrizada se crea una superclase de la que heredarán cada una de las clases con una parametrización específica. A esta clase se la puede referenciar utilizando el símbolo de interrogación como parámetro. Así, la clase `VectorDinamico<?>` es superclase de todas las clases `VectorDinamico`, y puede utilizarse para aquellos casos en los que deseemos realizar alguna operación sobre un `VectorDinamico` sin importarnos el tipo de objeto que almacena.

Hay que notar que un objeto de la clase `Concesionario<Deportivo>` y otro de la clase `Concesionario<Utilitario>` no comparten el mismo tipo, aunque las clases `Deportivo` y `Utilitario` estén en relación de herencia.

En los diagramas estáticos de *UML* las clases parametrizadas se dibujan con una pequeña caja en su parte superior derecha que denota su tipo.

Hay que notar que un objeto de la clase `VectorDinamico<A>` y otro de la clase `VectorDinamico<B>` no comparten el mismo tipo, aunque las clases `A` y `B` estén en relación de herencia. Así, un objeto de la clase `VectorDinamico<String>` no comparte el tipo con un objeto de la clase `VectorDinamico<Object>` y no se puede hacer el *casting* de un tipo a otro.

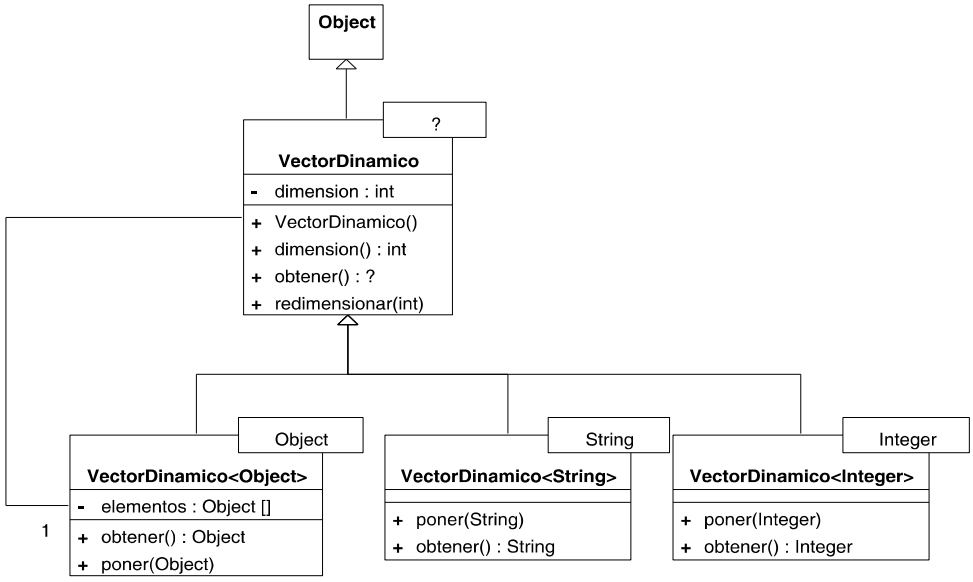


Figura 22.- Jerarquía de herencia entre clases parametrizadas.

El código para crear un objeto de la clase `VectorDinamico` y utilizarlo para almacenar cadenas con nombres sería:

```
VectorDinamico <String> v = new VectorDinamico <String>(10);
v.Poner(0,String("Juan"));
v.Poner(1,String("Andres"));
```

3.4.2 Métodos con tipos parametrizados

Es posible definir tipos genéricos en el interior de las clases y utilizarlos para definir los parámetros de los métodos y los valores devueltos por estos. Al definir el método, las llaves angulares aparecen después de los modificadores del método y antes del tipo devuelto.

```
public final <A extends String> int tamaño(A texto) {
   return texto.size();
}
```

Los tipos parametrizados suelen utilizarse sobre todo en los métodos estáticos, en los que por su naturaleza no puede accederse al tipo parametrizado de la clase.

3.5. Conclusiones

En este capítulo se ha visto que los objetos en *Java* solo pertenecen a una clase. Sin embargo el mismo objeto puede cumplir tantos tipos como clases e interfaces haya en la jerarquía de herencia de su clase. Esto, unido a la posibilidad de tratar a un objeto mediante una referencia de cualquiera de los tipos que cumple, da lugar al polimorfismo dinámico. También se ha estudiado la genericidad y la implementación que *Java* realiza de ella.

3.6. Lecturas recomendadas

"Piensa en *Java*", 2ª Edición, Bruce Eckel, Addison Wesley, 2002. En este libro, Eckel con un estilo desenfadado, presenta una obra de referencia imprescindible para profundizar en *Java*.

3.7. Ejercicios

Ejercicio 1

De la clase `A` se deriva la clase `B`. De la clase `B` se deriva la clase `C`. ¿Desde la clase `C` se puede llamar a los métodos protegidos de la clase `A`?

Ejercicio 2

¿Puede tener alguna utilidad una clase que no tiene propiedades y cuyos métodos son todos abstractos? En caso afirmativo explicar cuál sería.

Ejercicio 3

Las clases que implementen la interfaz `ElementoOrdenable` pueden determinar si un objeto es mayor que otro. Esta interfaz sólo tiene un método llamado `mayorQue()` que devuelve `true` si el objeto sobre el que se pregunta es mayor que el objeto que se pasa por parámetro. Es claro que esta relación de orden que define sobre una clase particular responde a la naturaleza singular de esa clase.

```
interface ElementoOrdenable {
   public boolean mayorQue(ElementoOrdenable e);
}
```

Se pide escribir el código *Java* correspondiente a un objeto que implemente un árbol binario ordenado de objetos `ElementoOrdenable`. Este árbol cumplirá la siguiente descripción:

Cada nodo de `ArbolBinario` podrá tener como máximo dos ramas descendentes (una a la izquierda y otra a la derecha) que serán a su vez cada uno objetos `ArbolBinario`.

La clase `ArbolBinario` permitirá navegar por sus ramas mediante dos funciones que devuelven el árbol binario de la izquierda o de la derecha. Estos métodos tendrán la forma:

```
public ArbolBinario izquierda();
public ArbolBinario derecha()
```

Se podrá obtener el objeto contenido en el nodo raíz de estos árboles mediante el método:

```
public ElementoOrdenable obtenerElementoRaiz();
```

La inserción de elementos en este árbol se realiza mediante el método:

```
public void insertar(ElementoOrdenable e);
```

La inserción sigue el siguiente algoritmo recursivo de dos pasos:

> Paso 1.- Si el árbol está vacío el elemento se inserta en la raíz.
>
> Paso 2.- Si el árbol no está vacío se compara el elemento a insertar con el presente en el nodo raíz. Si es mayor el nuevo elemento se intenta insertar en el subárbol que pende de la derecha del nodo raíz llamando al paso 1 con el subárbol de la derecha. En otro caso se intenta insertar en el subárbol que penda de la izquierda del nodo raíz, llamando al paso 1 con el subárbol de la izquierda.

Capítulo 4 El paquete java.lang

El paquete más importante de *Java* es `java.lang`. Este paquete aporta interfaces y clases tan fundamentales para *Java* que están integradas con el propio lenguaje y no es preciso importarlas. En este capítulo se estudiará la clase `Object` (clase base de toda clase creada en *Java*), se estudiarán las envolturas (recubrimientos de los tipos primitivos), se estudiará la clase `Exception` (que permitirá manejar errores) y se introducen las interfaces `Clonable` y `Comparable` y las clases de reflexión (que permiten obtener información sobre las propias clases en tiempo de ejecución).

4.1. La clase `Object`

Toda clase que se declara en *Java* y que no se especifica de qué clase deriva lo hace de la clase `Object`. Esto tiene como consecuencia que todas las clases de *Java* tienen como tipo común la clase `Object`.

Las ventajas de una jerarquía de raíz única son enormes. Entre ellas se pueden destacar dos:

- Todos los objetos en última instancia son del mismo tipo y por lo tanto puede garantizarse ciertas operaciones sobre todos ellos. Por ejemplo simplificó la creación del recolector de basura, ya que sólo hubo que implementar el comportamiento para la clase `Object`.
- Proporciona mucha flexibilidad a la hora de programar. Por ejemplo permite definir estructuras de datos que almacenen objetos de tipo `Object` (y por tanto cualquier clase de objetos, pues todas derivan de `Object`).

La clase `Object` define una serie de métodos que pueden utilizarse sobre todos los objetos que se creen en *Java*. Estos métodos son: `clone()`, `equals()`, `finalize()`, `getClass()`, `hashCode()`, `toString()`, `wait()`, `notify()` y `notifyAll()`.

El método `toString()` devuelve una cadena asociada al objeto. De esta forma todos los objetos en *Java* tienen una representación en forma de cadena de texto.

El método `hashCode()` devuelve un entero que puede utilizarse como identificador único del objeto en la máquina virtual.

De los métodos `wait()`, `notify()` y `notifyAll()` hablaremos cuando estudiemos la concurrencia.

Reflexión

Casi todos los métodos de `Object` utilizan los mecanismos de **reflexión** de *Java*. La reflexión es una propiedad muy potente que permite a un programa revisar su propio código de manera dinámica, e incluso permite añadir clases y miembros nuevos durante la fase de ejecución.

El método `getClass()` de `Object` es muy importante en este mecanismo de reflexión, pues devuelve un objeto de la clase `Class` que representa la clase a la que pertenece el objeto. La clase `Class` permite que un programa pueda realizar multitud de operaciones con clases desconocidas en tiempo de compilación. Así, esta clase permite inspeccionar los miembros de un objeto (`getMethod()`, `getFields()`, `isAbstract()`...), permite crear objetos pertenecientes a la clase que representa (usando `newInstance()`), e incluso permite ser cargada dinámicamente (mediante un objeto de tipo `ClassLoader`).

`clone()` es un método protegido que permite sacar una copia de un objeto. Esto es importante debido a que, recordemos, el operador de asignación sólo copia la referencia. Cuando la copia de un objeto no sólo consista en igualar los valores de sus propiedades se deberá redefinir el método `clone()` en la clase derivada, añadiendo el comportamiento necesario. La implementación de este método realizada en la clase `Object` solo es posible mediante mecanismos de reflexión, los cuales, obsérvese, violan la encapsulación, pues deben permitir acceder a partes privadas de la clase derivada para copiarlas.

El método `equals()` se proporciona para realizar la comparación de un objeto con otro. La implementación por defecto, proporcionada por `Object`, devuelve `true` sólo si se está comparando el objeto consigo mismo. Cuando se requiera que la comparación de dos objetos no se base simplemente en su identidad se deberá redefinir el método. En caso de que se opte por redefinir el método `equals` también deberá redefinirse el método `hash`, de tal modo que dos objetos devuelvan el mismo `hash` cuando al comprarlos con `equals` se devuelva `true`.

4.1.1 La clase `ClassLoader`

La clase `ClassLoader` crea la posibilidad de cargar dinámicamente una clase a partir de su código compilado. El siguiente fragmento de código carga de disco la clase que se le especifica por parámetros. Para ello se crea una clase que hereda de

`ClassLoader` y que implementa el método `loadClassData` de manera que crea desde disco un flujo de *bytes* que contiene los *bytecodes* de la clase a cargar.

```
import java.io.*;
import java.util.Vector;

public class Cargador extends ClassLoader
{
  public Class <?> findClass(String name) {
     byte[] b = loadClassData(name);
     return defineClass(name, b, 0, b.length);
  }

  private byte[] loadClassData(String name) {
    try {
      String nombre=".\\"+name+".class";
      InputStream in = new FileInputStream(nombre);
      Vector <Byte> buffer = new Vector<Byte>();

      int i = in.read();

      while(i != -1) {
        buffer.add((byte)i);
        i = in.read();
      }
      byte [] aux = new byte[buffer.size()];
      for (int cont = 0; cont < buffer.size(); cont++)
        aux[cont]=buffer.get(cont);
      return aux;
    }
    catch(Exception e) {
       return null;
    }
  }
}
```

El siguiente código carga la clase `Deportivo` que corresponde a una implementación de la clase `Coche`.

```
public static void main(String[] args) {
   try {
     ClassLoader loader = new Cargador();
     Class un_coche = loader.loadClass("Deportivo");
     Coche c = (Coche) un_coche.newInstance();
     c.acelerar();
   }
   catch(Exception e) {
     System.out.print(e.getMessage());
   }
}
```

4.2. Clases para el manejo de cadenas de caracteres

Java proporciona varias clases para facilitar la manipulación de cadenas caracteres frente al uso directo de *arrays* de caracteres. Estas clases son: `CharBuffer`, `Segment`, `String`, `StringBuilder` y `StringBuffer`.

Comparten todas ellas que implementan la interfaz `CharSequence`, la cual se caracteriza principalmente por ofrecer el método `charAt()` y `length()`.

Cada una de estas clases aporta ciertas ventajas de uso que desgraciadamente tienen un coste en eficiencia, por lo que no se puede elegir un enfoque como el menor, sino que es conveniente estudiar cuál es la clase más adecuada para cada caso.

La clase `CharBuffer` es la más eficiente tras el *arrays* de caracteres, pero también es la que menos métodos de tratamiento aporta.

La clase `String` es la más conocida debido a que todos los objetos poseen un método `toString()` proporciona multitud de métodos para facilitar el tratamiento de cadenas de caracteres: `substring()`, `trim()`, `compareTo()`, `toCharArray()`, `valueOf()`. Los arrays de caracteres respecto a la clase `String` disponen de las propiedades ***autoboxing*** y ***autounboxing***, de manera que el compilador convierte un *array* de caracteres en un `String` y un `String` en un *array* de caracteres automáticamente. Además, se han sobrecargado los operadores + y += para permitir una notación simple para la concatenación de un `String` con otro o con cualquier tipo primitivo.

Además la clase `String` proporciona métodos estáticos (como `valueOf()`) que permiten realizar operaciones sin tener que crear objetos de la clase `String`.

Por otro lado hay que tener en cuenta que cualquier cadena entre comillas es convertida en un `String` automáticamente por el compilador, de manera que no se pueden comparar dos objetos `String` usando el operador == pues se está comparando la referencia a un objeto `String` y no el objeto en si mismo.

```
String texto = "Un array de caracteres se convierte en String";
String numero = cadena + "Ejemplo";    //El operador + está sobrecargado
   numero += 5;                        //El operador += convierte el 5 en cadena
```

Una característica importante de la clase `String` es que es **inmutable**, es decir, un `String` no se puede modificar. Así, añadir un carácter a un `String` implica la creación de un `String` nuevo. Este comportamiento es adecuado en algunos casos, pero muy poco eficiente cuando por ejemplo se desea construir una cadena mediante la adición de caracteres. Por ello *Java* proporciona las clases `StringBuilder` y `StringBuffer`. Estas clases no son inmutables y por tanto sí permiten la adición de caracteres. La diferencia entre ambas consiste en que `StringBuilder` no es resistente a colisiones cuando se utiliza desde múltiples hilos simultáneamente, mientras que `StringBuffer` sí lo es, a coste de una menor eficiencia.

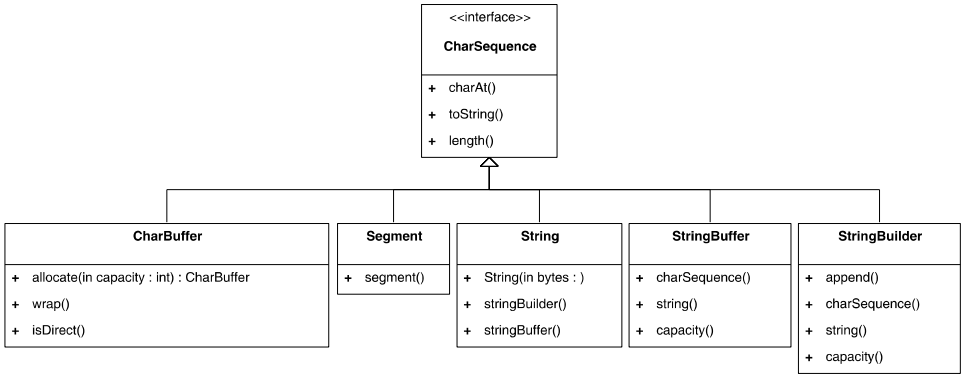


Figura 23.- Jerarquía de clases de usadas para manipulación de cadenas de texto.

Finalmente, la clase `Segment` se utiliza para manipular segmentos de otras cadenas de caracteres sin necesidad de copiarlos en un nuevo objeto.

4.3. Las excepciones

El manejo de errores siempre ha constituido un aspecto difícil de abordar. La mayoría de esquemas de tratamiento de errores dependen de la disciplina del programador y se basan en la devolución de valores o en el uso de elementos comunes a todo el código (variables o ficheros comunes, como el `stderr` de C). Si no se sigue de forma rigurosa una estrategia bien definida respecto al tratamiento de errores el resultado suele ser un programa inestable debido a situaciones de error no contempladas.

En los lenguajes de programación orientados a objetos el concepto de error se cambia por el de la situación excepcional que se produce cuando un objeto no puede cumplir su contrato. Una **excepción** es un objeto que se lanza en el momento en el que ocurre una situación excepcional y que se captura cuando esa situación excepcional puede tratarse (reintentando, realizando una acción alternativa o interrumpiendo la ejecución del programa).

En *Java*, para lanzar una excepción se usa la palabra reservada `throw`, y el objeto que se lanza debe ser heredero de la clase `Throwable` o de alguna subclase de ésta. Al lanzar la excepción se interrumpe la secuencia normal de ejecución, recorriéndose la pila de llamadas hasta que se encuentra un ámbito en el que se capture ese tipo de excepción. Por ejemplo, cuando el siguiente fragmento de código detecta que podría producirse una división por cero lanza una excepción.

```
...
int obtenerRatioDeConsumo () {
   if (kilometrosRecorridos == 0)
      throw new ArithmeticException ();
   return litros / kilometrosRecorridos;
}
```

Para definir el ámbito en el que se captura una excepción y el ámbito de su tratamiento se utiliza la una estructura basada en los *tokens* `try-catch-finally` que se expone a continuación.

```
<bloque try> ::= try '{' [<Código que puede producir la excepción>] '}'
[catch'('<tipo de excepción>')' '{' [<Código que trata la excepción>] '}']+
[finally '{' [<Código que se ejecuta tras el try o tras los catch>] '}']
```

El bloque `try` contiene el fragmento de código que se "intenta" ejecutar. La idea consiste en que si la ejecución de ese código, o de algún método ejecutado por ese código, a cualquier nivel, genera una excepción, pueda ser capturada en el bloque `catch` correspondiente a su tipo.

Una excepción del tipo `T` sólo puede ser capturada por un gestor del tipo `T` o de un tipo ascendiente de `T`. En otras palabras, un bloque `catch` captura las excepciones del tipo que declara y de los tipos que de él deriven. Este comportamiento polimórfico hace que los `catch` deban situarse de más específicos a más generales respecto al tipo que declaren. En otro caso, nunca recibirán entradas, al estar eclipsados por tipos más generales.

Una vez que un bloque `catch` captura una excepción, la ejecución continúa de manera normal a no ser que se lance otra excepción. Es decir, la situación excepcional desaparece. Por ello los bloques `catch` deben situarse donde sean útiles para gestionar los errores que capturan y no antes.

El bloque `finally` contiene código que se desea que siempre se ejecute antes de abandonar el bloque `try/catch`. Esto significa que el bloque `finally` se ejecuta se produzca excepción o no se produzca, y se capture o no se capture. El bloque `finally` se ejecutará incluso si en alguno de los bloques se ejecuta una instrucción `return`.

Para el ejemplo anterior el tratamiento de excepciones quedaría como sigue:

```
,,,
try {
  obtenerRatioDeConsumo();
} catch(ArithmeticException e) {
    tratamientoDeErrorPorExcepcionAritmetica(e);
} catch(Throwable t) {
    tratamientoDeErrorPorCualquierOtraExcepcion(t);
} finally {
    codigoQueSiempreSeEjecuta();
}
```

4.3.1 Tipos de excepciones

En *Java* hay dos tipos de excepciones: **controladas** (*checked exceptions*) y **no controladas** (*unchecked exceptions*). Si existe la posibilidad de que un método lance excepciones controladas debe indicarlo obligatoriamente en su definición utilizando la palabra reservada throws seguida de los tipos de objetos que puede lanzar. Además, si un método `M` llama a otro que puede lanzar cierta excepción `E`, `M` debe tratar esa excepción o avisar de que él también puede lanzar la excepción `E`. Los métodos que lancen excepciones no controladas no tienen porqué declarar que lanzan ningún tipo de excepción.

Derivada de `Throwable` hay dos clases: `Error` y `Exception`. A su vez, derivada de `Exception` encontramos la clase `RuntimeException`. Cualquier clase que derive de `Error` o de `RuntimeException` podrá ser utilizada para generar excepciones no controladas. Por otro lado, las clases que deriven directamente de Exception permitirán crear excepciones controladas.

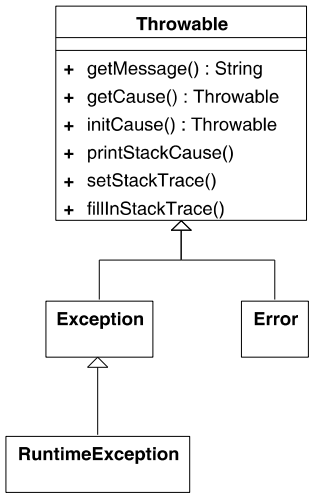


Figura 24.- Jerarquía de clases de excepción.

Una excepción no es un valor devuelto por una función. Esto hace que el tratamiento de errores se simplifique respecto al método de devolución de valores, pues no es necesario ir comprobando tras cada llamada a cada método si todo va bien o no. Además, una excepción no puede ser ignorada, lo que garantiza que la situación excepcional será tratada antes o después, y que posiblemente será restablecida la ejecución normal del programa.

Ejemplo de excepciones en el Parking

Si se examina el método `desocuparPlaza()`, en el ejemplo de la clase `Parking` del punto 2.3.10, se comprobará que cuando no pueda cumplir su contrato, por estar el objeto `Parking` vacío, se lanzará una excepción de tipo `nullPointerException`. Si se desease que el programador, usuario de la clase `Parking`, obtuviese una explicación más detallada del problema se podría utilizarse una excepción específica. El siguiente fragmento de código define una clase derivada de `RuntimeException` para ejemplificar este caso.

```
import java.util.*;

class ErrorParking extends RuntimeException {
  public String getMessage() {
     return "El parking está vacío";
  }
}
```

A continuación se presenta como debería ser el método `desocuparPlaza()` para que lanzase excepciones `ErrorParking` cuando le sea imposible desarrollar su cometido.

```
 class Parking {
    ...
    public void desocuparPlaza() throws ErrorParking{
           if (plazaActual == null) {
             throw new ErrorParking();
           }
           ...
       }
     ...
 }
```

Finalmente se presenta el código de un programa que usa la clase `Parking`.

```
class ParkingMain {
    public static void main (String [ ] arg) {
       Parking parking = new Parking();
       parking.ocuparPlaza();

       try {
          parking.desocuparPlaza();
       } catch (ErrorParking e) {
          System.out.println (e.getMessage());
       }
    }
 }
```

4.3.2 El uso práctico de las excepciones

Las excepciones deben utilizarse con precaución para evitar que su uso se transforme en algo peor que lo que se intenta simplificar, es decir peor que tratar los errores mediante el uso de los valores devueltos por los métodos. A continuación se exponen algunos criterios que deben seguirse:

- **Limitar el uso de excepciones controladas.** La excepciones controladas deben utilizarse solo cuando su tratamiento por un tercero implique tomar una decisión o que como consecuencia de la excepción se deba realizar algún tratamiento lógico imprescindible. En el resto de casos deben utilizarse excepciones no controladas (derivadas de `RuntimeException`). En particular no deben utilizarse excepciones controladas para expresar errores relativos al mal uso de una clase por parte del programador. La excepción `IOException` lanzada por `read` cuando se produce un error en la lectura de un disco debe ser controlada, pues un fallo de lectura es algo que, aunque inesperado, no implica un fallo de programación y siempre necesita un tratamiento que no debe omitirse. Por otro lado, la excepción `ArrayIndexOutOfBounds` es una excepción no controlada enviada por *Java* cuando se intenta acceder más allá del límite de un *array*. Si la excepción fuese controlada cada método que usase un *array* tendría que tratarla o avisar que la lanza, cosa inútil si tenemos en cuenta que un programa bien escrito nunca debería sobrepasar el límite de un *array*. Obsérvese que, cumpliendo este criterio, en el ejemplo de la clase `Parking` que se ha propuesto en el punto anterior, la clase `ErrorParking` deriva de `RuntimeExcepcion`.
- **Utilizar excepciones de *Java*.-** Cuando el tipo de error de una aplicación sea similar a un error definido en el *API* de *Java* debe usarse la clase de *Java*. De esta forma se facilita al usuario de la clase la compresión del error. Entre las excepciones más comunes del *API* de *Java* podemos destacar: `IllegalArgumentException`, `ArithmeticException`, `SecurityException`, `NullPointerException`, `IllegalStateException`, `IllegalArgumentException` y `IndexOutOf-BoundsException`.

- **Cambiar el nivel de abstracción de las excepciones.** Las excepciones lanzadas por un método deben estar al mismo nivel de abstracción que la clase que implementa el método. Por ejemplo, la clase `Tabla` de un hipotético paquete `BaseDeDatos` no debe lanzar excepciones de tipo `IOException` cuando falle una operación interna contra un fichero auxiliar. Tal error se debería a un detalle interno de implementación y para fomentar la encapsulación no debería traslucir. En vez de esto debería capturarse la `IOException` y definirse una clase especifica para las excepciones de la base de datos que sería la que se lanzaría. No seguir esta norma haría que los usuarios de la base de datos pudiesen no entender los mensajes de error.
- **No lanzar excepciones innecesarias.** El enfoque de programación defensiva consiste en comprobar los parámetros que se reciben y en caso de ser incorrectos lanzar excepciones. Este enfoque es erróneo por dos razones: primero, cuando el programa esté bien construido esas comprobaciones serán innecesarias, y segundo, el usuario del método incorrectamente invocado no puede hace nada con el error, sólo corregir el programa. Según esta recomendación, quizás no fue tan buena la idea de definir la excepción `ErrorParking` en el ejemplo anterior, ya que el código anterior ya producía `NullPointerException`.
- **Encadenar los mensajes de las excepciones.** Las excepciones no controladas que se generen deben contener información suficiente para que el programador pueda trazar el error. Para ello, cuando se captura una excepción para cambiar su nivel de abstracción, se debe añadir, al mensaje de la excepción de alto nivel, la información de la excepción de bajo nivel que le dio origen.

Ejemplo del uso de diferentes tipos de excepciones

En el siguiente ejemplo se utilizan los dos tipos de excepciones. La excepción `ExcepcionPorNoGasolina` de tipo controlada y las `ExcepcionPorNoArrancado` e `IllegalArgumentException` no controladas. Obsérvese que mientras que la primera obligatoriamente debe ser indicada y capturada, las siguientes no.

```
/**
* Clase coche
* @version 7.0
*/
public class Coche {
  private boolean motorEncendido;       //Indica si el motor está encendido
  private int velocidad;                //Indica la velocidad del coche
  private int litrosGasolina;

   /**
    * Constructor de coche con el motor apagado y velocidad nula
    * @param litrosGasolina Litros de gasolina con los que se inicia el coche
    */
  public Coche(final int litrosGasolina) {
      this.litrosGasolina = litrosGasolina;
  }
   /**
    * Enciende el motor del coche.
    * @throws ExcepcionPorQuedareSinGasolina Si nos hemos quedado sin gasolina
    */
  void arrancar() throws ExcepcionPorNoGasolina {
      //Comprobamos que tenemos gasolina
      if (litrosGasolina == 0) {
          throw new ExcepcionPorNoGasolina();
      }
      //Encendemos el motor
      motorEncendido = true;
  }
  /**
   * Acelera el coche, aunque se impiden valores superiores a 140
   * @throws ExcepcionPorNoEstarArrancado Si no hemos arrancado el coche
   *          se lanza esta excepción.
   *          Como es runtime no se declara en el throws.
   * @throws IllegalArgumentException No se puede pasar de 140
   */
  void acelerar(final int incremento) {

      //Comprobamos que el motor está encendido
      if (!motorEncendido) {
          throw new ExcepcionPorNoArrancado();
      }

       if (velocidad + incremento > 140) {
          throw new IllegalArgumentException();
      }

      velocidad += incremento;  //Aumentamos la velocidad
  }
}
```

El código para las excepciones sería:

```
public class ExcepcionPorNoGasolina extends Exception{
    //Como deriva de Exception se debe declarar su lanzamiento
    //Además, donde se use es obligatoria su captura o la delaración de lanzamiento
}

public class ExcepcionPorNoArrancado extends RuntimeException{
}
```

El código de `main` para este ejemplo es:

```
package libro.ejemplos.cuatro.tres.uno;

public class CocheMain {
    public static void main(String[] args) {
        final Coche coche = new Coche(0);        //Creamos un coche sin gasolina
        try {
            coche.arrancar();  //intentamos arrancar el coche sin gasolina
        } catch (ExcepcionPorNoGasolina e) {
             System.out.println("¡Nos hemos quedado sin gasolina!");
        }
        try {
            coche.acelerar();  //intentamos acelerar el coche sin haberlo arrancado
        } catch (ExcepcionPorNoArrancado e) {
            System.out.println("¡El motor no está arrancado!");
        } catch (Exception e) {                  //Esta captura debe ser la última
            System.out.println("¡Algún otro error!"); //en otro caso lo captura todo
     }
    }
}
```

Y la salida obtenida por este programa será:

```
java CocheMain
¡Nos hemos quedado sin gasolina!
¡El motor no está arrancado!
```

Problemática de las excepciones controladas

Las excepciones controladas han creado mucha controversia en el mundo del desarrollo de *software*. Cuando aparecieron como parte del lenguaje *Java* prácticamente toda la comunidad estaba de acuerdo en lo útil y necesarias que eran para un buen tratamiento de los errores. Parecía una forma sencilla de hacer que las clases fueran usadas de forma correcta incluso cuando lanzaran excepciones. Sin embargo, ese sentimiento de "esto es lo que yo necesitaba" provocó que se abusara de ellas, ocasionando justo lo contrario de lo que se quería promover.

Obligar al tratamiento de una excepción, mediante las excepciones controladas, en muchas ocasiones se convierte en algo engorroso y poco útil, ya que el usuario de dicha clase no puede hacer nada para solucionar dicha excepción salvo relanzarla a un nivel superior. Cuando esto ocurre las excepciones controladas solo sirven para oscurecer el código fuente con innecesarios bloques `try` y `catch` que dificultan su compresión y mantenimiento.

Por ello, muchos opinan que las excepciones controladas no son necesarias ya que, además de los problemas comentados, ningún otro lenguaje las ha vuelto a implementar. Solo unos pocos aún las defienden, aunque puntualizando que no debe abusarse de ellas como se ha hecho en ocasiones. En general se recomienda no utilizar excepciones controladas, ya que el uso de excepciones no controladas es suficiente para manejar los errores y no aparecen los problemas descritos.

4.4. Sobrecarga de los métodos básicos de `object`

Los siguientes puntos tratan en detalle algunos de los métodos más importantes de *Java* que suelen sobrecargarse para dar una funcionalidad adecuada a las clases que se construyen.

4.4.1 Interfaz `Cloneable`

Ya se ha dicho que cuando en *Java* se utiliza el operador de asignación entre dos referencias, el objeto involucrado no se copia, sino que se obtienen dos referencias al mismo objeto. En algunas situaciones esto puede no ser deseable. Por ejemplo, imaginemos cuando se pasa un objeto como parámetro a un método. Si el método modifica el objeto, esta modificación afectará también al objeto del invocador del método porque en realidad es el mismo. Si fuese necesario evitar esto se debería realizar una copia de tal objeto.

La alternativa clásica para resolver este problema consiste en la implementación del constructor de copia. Éste es un constructor que recibe como parámetro un objeto de la propia clase, para construir a partir de él una copia. Sin embargo,

esta solución no es válida cuando se desea que una interfaz refleje la posibilidad de copiar (recordemos que en las interfaces no se pueden declarar constructores).

Para hacer que este tipo de copias se realicen con un formato común, *Java* proporciona en la clase `Object` el método protegido `clone()`. El método se declara como `protected` para impedir que se clonen objetos si el diseñador de una clase no lo desea. Por eso, si se desea que se puedan clonar objetos es preciso hacer público un método que lo invoque. La mayoría de las clases del *API* de *Java* lo tienen público.

El método `clone()` implementado en la clase `Object` utiliza la reflexión para copiar los valores de las propiedades del objeto. Así, las referencias de un objeto clonado apuntan a los mismos objetos que el original. Esto es, por defecto al clonar un objeto no se realiza un clonado recursivo de los objetos en él contenidos. Cuando no se desea este comportamiento es preciso sobrecargar `clone()`, y programar explícitamente que se clonen los objetos referenciados por las propiedades.

Por defecto no es posible clonar los objetos de una clase. Para empezar se debe declarar el método como `public`, ya que en `Object` es `protected`. Además, se debe indicar que dicha clase implementa la interfaz `Cloneable`. Esta interfaz, cuya declaración está vacía, se utiliza para indicar que sobre esa clase está permitido invocar al método `clone()`, en otro caso *Java* lanzará una excepción del tipo `CloneNotSupportedException`.

El siguiente ejemplo hace público un método para clonar objetos de la clase `Coche`. La propiedad velocidad se copia al invocar a `clone()` de `Object`, pero el objeto `Rueda` no se clona, sino que simplemente se copia su referencia. Por eso, en el ejemplo se añade el código preciso para que se clone también el objeto rueda.

```
class Rueda implements Cloneable {
    public Rueda clone() throws CloneNotSupportedException {
        return (Rueda) super.clone();
    }
}

class Coche implements Cloneable {
    public int velocidad;

    public Rueda r = new Rueda();

    public Coche clone()  throws CloneNotSupportedException {
        Coche c = (Coche) super.clone();
        c.r = r.clone();
        return c;
    }
}
```

Clone presenta una serie de problemas que deben tenerse en cuenta. Para empezar hay recordar que si `clone` no se implementa correctamente en una clase base, las clases derivadas serán incapaces de copiar correctamente aquellas propiedades privadas que requieran un tratamiento especial. Además, `clone` no podrá modificar las propiedades finales de ningún objeto clonado, pues las propiedades finales solo se pueden modificar en los constructores. Por todo esto, se recomienda utilizar un constructor copia, y sólo cuando sea imprescindible implementar `clone` haciendo que invoque a dicho constructor.

4.4.2 Los métodos `equals` y `hashCode`

Ya se ha comentado que, en su implementación por defecto, `equals()` solo devuelve `true` si se compara un objeto consigo mismo. Esta implementación del método `equals()`, por parte de *Java*, puede resultar insuficiente en muchos usos prácticos. Por ejemplo, puede ser deseable que la comparación de dos objetos clonados devuelva `true` (por defecto, la comparación de dos objetos clonados devolverá `false`, debido a que son dos instancias diferentes).

Para que las bibliotecas de *Java* funcionen correctamente sobre una clase que redefine `equals()`, dicha implementación debe cumplir las siguientes propiedades:

- Identidad.- `a.equals(a)` debe devolver `true`, para todo `a`.
- Simetría.- `a.equals(b)` si y solo si `b.equals(a)`, para todo `a y b`.
- Transitividad.- Si `a.equals(b)` y `b.equals(c)` entonces `a.equals(c)`, para todo `a b` y `c`.
- Elemento neutro.- `a.equals(null)` debe devolver `false`, para todo `a`.
- Inmutabilidad.- Si en algún momento se cumple `a.equals(b)` o `!a.equals(b)` su valor no puede cambiar.

Además, toda sobrecarga del método `equals()` debe ir acompañada de una reimplementación del método `hashCode()` sobre las mismas propiedades, de manera que si `equals` devuelve `true` para dos objetos, `hashCode()` devuelva un valor idéntico al ser invocado sobre esos objetos. El siguiente fragmento de código muestra un ejemplo donde se aprecian estos detalles.

```
class Coche {
   private final int matricula;

   public Coche (int matricula) {
      this.matricula = matricula;
   }

   public boolean equals(Object o) {
      if (!o instanceof Coche)        //Si o es null instanceof devuelve false
         return false;
      return ((Coche)o).matricula == matricula;
   }

   public int hashCode() {
      return matricula;
   }
}
```

4.4.3 Interfaz `Comparable`

La interfaz Comparable, que también se encuentra en el paquete `java.lang`, permite realizar comparación entre los objetos de aquellas clases que la implementan. La interfaz `Comparable` tiene sólo un método: `int compareTo(Object)`. Este método tiene por objeto devolver un valor igual a cero si el resultado de la comparación es la igualdad, un valor negativo si el objeto que recibe el mensaje es menor que el que se le pasa como argumento, y un valor positivo si es mayor.

```
class Coche implements Comparable {
   private final int matricula;

   public int compareTo(Object o) {
      Coche c = (Coche) o;
      if (this. matricula == c.matricula )
         return 0;
      else if (this. matricula > c.matricula )
         return 1;
      else
         return -1;
   }
}
```

Todas las clases definidas en los paquetes estándar de *Java* suelen implementar esta interfaz. Por ejemplo la clase `String` cumple la interfaz `Comparable`.

Al igual que en `equals()`, es importante que la comparación se base en propiedades inmutables del objeto, en otro caso, una lista ordenada de objetos podría dejar de estarlo como consecuencia del cambio de una propiedad de uno de los objetos contenidos.

4.5. La concurrencia

Java aporta varios elementos que facilitan la concurrencia en los programas. El principal consiste en que para crear un hilo separado de ejecución (o *thread*) basta con heredar de la clase `Thread` e implementar el método `run()` que en la clase `Thread` es abstracto.

Una vez ejecutado el constructor de un objeto de una clase derivada de la clase `Thread` se debe invocar al método `start()` para iniciar su ejecución. Tras esta llamada, la máquina virtual de *Java* llama al método `run()` y este comienza su ejecución en el nuevo *thread*.

Si no se desea heredar de la clase `Thread` también es posible implementar la interfaz `Runnable`, aunque en este caso el *thread* será iniciado inmediatamente tras la creación del objeto llamándose al método `run()`.

En programación concurrente se suelen producir diferentes tipos de problemas. Podemos citar tres principales:

- Las **condiciones de carrera**, que ocurren cuando dos hilos acceden a un recurso simultáneamente y como consecuencia el resultado del acceso es indeterminado.
- Los **bloqueos mutuos** (o ***deadlocks***), que paralizan la ejecución de un programa, debido a que los hilos quedan esperando sucesos que nunca se producirán, porque dependen unos de otros y están todos esperando.
- La **inanición** (o ***starvation***), que ocurre cuando un hilo se queda esperando un suceso que puede no ocurrir o que ocurre poco frecuentemente.

Para evitar estos problemas, y para facilitar el desarrollo de programas concurrentes, *Java* aporta varios mecanismos. Así, *Java* dispone de mecanismos para marcar regiones de exclusión mutua (en las que solo un hilo puede estar activo) y paso de

mensajes (para que los hilos se comuniquen y eviten los problemas). Además, el paquete `java.util.concurrent`, añade sobre estos mecanismos otros más sofisticados (como semáforos, colas bloqueantes o contadores).

Además, muchas clases de *Java* son inmutables, como por ejemplo `String`. El uso de clases inmutables evita muchos problemas en ámbitos concurrentes, pues como los objetos no pueden cambiar, no se pueden producir condiciones de carrera en ellos.

4.5.1 La palabra reservada `synchronized`

Este mecanismo permite asegurar que dos hilos de ejecución no podrán penetrar a la vez en cierta región de un objeto, creando lo que se conoce como una **región de exclusión mutua**.

El siguiente código ilustra como definir un ámbito que contenga la parte donde se puedan producir colisiones.

```
class Coche {
    int velocidad;
    public void acelerar(int v) {
        if (v > 0)
            synchronized  {
                velocidad += v;
            }
    }
}
```

La palabra reservada `synchronized` se puede utilizar también para crear monitores. Un **monitor** permite la ejecución en exclusión mutua de los métodos o ámbitos etiquetados como `synchronized` dentro de una instancia de una clase. El hilo que no pueda entrar en una región marcada por `synchronized` esperará a que el otro hilo abandone la región. Así, *Java* permite dotar a la parte estática y a cada objeto de una clase con un monitor diferente.

Así, como la siguiente clase especifica `acelerar()` como `synchronized`, se puede invocar dicho método desde dos hilos de ejecución sin que se produzcan colisiones.

```
class Coche {
    int velocidad;
    public synchronized void acelerar(int v) {
        velocidad += v;
    }
}
```

La palabra reservada `synchronized` también se puede utilizar al crear un ámbito, pasándole un objeto sobre el que realizar la sincronización. En este caso para iniciar un bloque de código que se desea que se ejecute en exclusión mutua. Esta exclusión mutua se realiza a nivel de la instancia del objeto que la ejecuta. El siguiente fragmento de código ilustra este uso sobre un objeto de la clase `Aparcamiento` del capítulo 2.

```
class Coche {
    Aparcamiento aparcamiento = new  Aparcamiento();
    public void aparcar() {
        synchronized  (aparcamiento){
            aparcamiento.Aparcar(this);
        }
    }
}
```

4.5.2 Comunicación entre hilos

Java define cuatro valores diferentes para definir el estado de un *thread* dentro de un objeto o de la parte estática de una clase: **nuevo**, **ejecutable**, **muerto** y **bloqueado**.

Un hilo está en estado nuevo cuando acaba de iniciarse y aún no ha comenzado a ejecutarse. Está en estado ejecutable cuando se está ejecutando o cuando no lo está pero nada impide que estuviese ejecutándose. Y está muerto cuando ha finalizado su método `run`.

Un *thread* en un objeto está bloqueado si:

- Ha llamado a `sleep()`, indicando el lapso de tiempo que duerme el hilo.
- Si se está esperando por algún evento de Entrada/Salida.
- Si está esperando para ejecutar código etiquetado como `synchronized` y aún no ha podido hacerlo.

- Y finalmente, si se ha llamado a `wait()` y está esperando que otro *thread* realice una llamada al método `notify()` del objeto. Este caso es el que permite la comunicación entre hilos, siendo en realidad la llamada a `notify()` un mensaje que envía un hilo a otro (o a otros si se usa `notifyAll()`).

Finalmente cabe decir que *Java* también permite asignar a los *threads* un nivel de prioridad mediante los métodos `getPriority()` y `setPriority()`. Así, el planificador de la Máquina Virtual de *Java* dispone de un criterio con el que distribuir el tiempo de ejecución entre los diferentes *threads*.

4.6. Los tipos enumerados

Los tipos enumerados también han sido añadidos en la versión 5 de *Java*. Antes, la definición de tipos enumerados no estaba soportada en *Java*.

Como sustituto simple, algunos programadores utilizaban propiedades enteras estáticas y finales. Esta solución tenía el problema de que no aseguraba el tipado, permitiendo mezclar estos primitivos enumerados con enteros. El siguiente ejemplo muestra una definición típica sobre *Java* 1.4 para el enumerado `Dia`.

```
class Dia {
   public final static int lunes      = 0;
   public final static int martes     = 1;
   public final static int miercoles  = 2;
   public final static int jueves     = 3;
   public final static int viernes    = 4;
   public final static int sabado     = 5;
   public final static int domingo    = 6;
}
```

Como se ve en el siguiente ejemplo, esta solución equivale en capacidad a los enumerados de *C* y *C++*.

```
int d1 = Dia.lunes;
int d2 = Dia.martes;
boolean posterior = (d2 > d1);
```

La sintaxis de estos enumerados resultaba compleja. Pero, lo peor es que potenciaban ciertos efectos indeseables, como mezclar el valor entero del enumerado con su semántica. Por ejemplo, al sumar 1 al `lunes` obtenemos el `martes`, pero al sumar 1 al `domingo` obtenemos un valor fuera del enumerado. Este problema se deriva de la mezcla de tipos.

Otra posibilidad, más compleja, consistía en utilizar una clase de la que sólo se creaban un número limitado de objetos, que correspondían a los valores enumerados (patrón de diseño *Enum*). Para hacer esto se necesita que el constructor sea privado y que se creen estáticamente los objetos como parte de la clase. El siguiente fragmento de código muestra el caso.

```
class Dia {
   private Dia() {}
   public final static Dia lunes      = new Dia();
   public final static Dia martes     = new Dia();
   public final static Dia miercoles  = new Dia();
   public final static Dia jueves     = new Dia();
   public final static Dia viernes    = new Dia();
   public final static Dia sabado     = new Dia();
   public final static Dia sabado     = new Dia();
}
```

Esta solución resuelve el problema del tipado. Además, permite añadir métodos que pueden ser convenientes para comparar elementos del enumerado o para describir sus propiedades. Sin embargo, la definición sigue siendo demasiado compleja.

La sintaxis introducida con la versión 5 permite definir un tipo enumerado con una sintaxis similar a la de *C++*. Mientras que, internamente, se crea una estructura similar a la comentada.

```
enum <Identificador del tipo> {[identificadores]*};
```

El siguiente ejemplo define el tipo `Dia` y lo usa.

```
enum Dia {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO};
Dia d = LUNES;
```

Además, como complemento a la introducción de los tipos enumerados la instrucción `switch` se amplió para permitir su uso sobre ellos y no solo sobre expresiones aritméticas enteras.

En la implementación de `enum` en *Java* lo que estamos haciendo en realidad es definir una clase y por ello un enumerado puede tener métodos y propiedades. De hecho, como todas las clases en *Java*, el tipo `enum` hereda de la clase `Object` y por ello ya trae de serie varios métodos. Esto hace que los enumerados en *Java* sean muy potentes, ya que permiten que se le puedan añadir comportamiento como a cualquier otra clase. Veamos el ejemplo del enumerado de los días de la semana pero con algo más de comportamiento.

```
enum DiasSemana {
    LUNES("Lunes"),
    MARTES("Martes"),
    MIERCOLES("Miercoles"),
    JUEVES("Jueves"),
    VIERNES("Viernes"),
    SABADO("Sabado"),
    DOMINGO("Domingo");

    private final String nombre;
    private DiasSemana(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public String toString() {
        return nombre;
    }

    public DiasSemana siguiente() {
        switch(this) {
            case LUNES         : return MARTES;
            case MARTES        : return MIERCOLES;
            case MIERCOLES     : return JUEVES;
            case JUEVES        : return VIERNES;
            case VIERNES       : return SABADO;
            case SABADO        : return DOMINGO;
            case DOMINGO       : return LUNES;
            default            : throw new IllegalStateException();
        }
    }
}
```

Si observamos las diferencias con la anterior implementación podemos ver que se han añadido varias funcionalidades al enumerado. Para empezar, se ha creado un constructor para guardar la cadena con la que se desea que se escriba dicho enumerado al invocar `toString` (recordemos que `enum` hereda de `Object` por lo que podemos sobreescribir `toString`). Además, se ha añadido un método a la clase para que devuelva el día siguiente al actual.

```
boolean iguales = MIERCOLES.siguiente() == JUEVES;
```

Tal y como se ha visto, se pueden definir constructores para los enumerados de forma que cada objeto del enumerado guarde una propiedad que lo defina. Una muestra de la potencia de los enumerados en *Java* está en que dicha variable puede a su vez ser un enumerado de otro tipo. Veamos un ejemplo práctico.

```
enum TipoDia {
    LABORABLE,
    NO_LABORABLE;
}
enum DiasSemanaClasificados {
    LUNES("Lunes",            TipoDia.LABORABLE),
    MARTES("Martes",          TipoDia.LABORABLE),
    MIERCOLES("Miercoles",    TipoDia.LABORABLE),
    JUEVES("Jueves",          TipoDia.LABORABLE),
    VIERNES("Viernes",        TipoDia.LABORABLE),
    SABADO("Sabado",          TipoDia.NO_LABORABLE),
    DOMINGO("Domingo",        TipoDia.NO_LABORABLE);

    private final String nombre;
    private final TipoDia tipoDia;

    private DiasSemanaClasificados(String nombre, TipoDia tipoDia) {
        this.nombre = nombre;
        this.tipoDia = tipoDia;
    }

    @Override
    public String toString() {
        return nombre;
    }
```

```
    public DiasSemanaClasificados siguiente() {
        switch(this) {
            case LUNES         : return MARTES;
            case MARTES        : return MIERCOLES;
            case MIERCOLES     : return JUEVES;
            case JUEVES        : return VIERNES;
            case VIERNES       : return SABADO;
            case SABADO        : return DOMINGO;
            case DOMINGO       : return LUNES;
            default            : throw new IllegalStateException();
        }
    }

    public boolean soyLaborable() {
        return tipoDia.equals(TipoDia.LABORABLE);
    }
}
```

Aunque, tal y como hemos visto, los enumerados son una herramienta muy potente, no es bueno abusar de ellos, ya que son objetos estáticos que dificultan la prueba de los componentes que los usan. Se deben usar enumerados cuando existe un grupo finito y conocido de objetos que no van a modificar su estado nunca (los días de la semana o los planetas del sistema solar son un conjunto finito y conocido).

4.7. Envolturas

Ya se ha comentado que los tipos primitivos en *Java* no son clases. Esta particularidad ha sido criticada por los puristas de la Programación Orientada a Objetos y alabada con la misma intensidad por los programadores preocupados por la eficiencia del código.

Para soslayar este aspecto, *Java* introduce las **clases de envoltura**. Estas clases recubren cada uno de los tipos primitivos en una clase, para que se puedan utilizar como objetos. Además, asociado a cada una de estas clases, *Java* proporciona un conjunto de métodos que facilitan las transformaciones y el resto de operaciones que con frecuencia es preciso realizar con los tipos primitivos (como conversiones entre tipos o conversiones a texto).

Obsérvese que como la definición de los genéricos en *Java* impide su uso sobre los tipos primitivos, los envoltorios pasan a ser imprescindibles.

El diagrama de la Figura 25 muestra las relaciones de herencia entre las diferentes clases de envoltura y algunos de sus métodos.

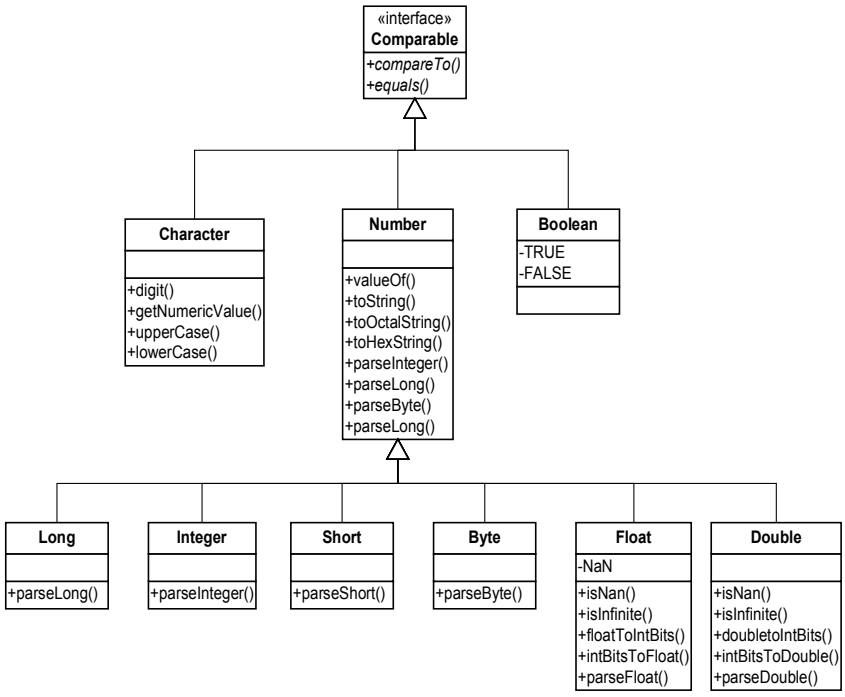


Figura 25.- Diagrama con los envoltorios de los tipos primitivos.

A partir de la versión 1.5, y al igual que ocurría con la clase `String` y los *arrays* de caracteres (ver capítulo 2), a la relación entre los tipos primitivos y las clases de envoltura se les ha dotado de la propiedad de *autoboxing* y *autounboxing*. De esta

forma es posible crear un objeto de una clase de envoltura utilizando directamente una asignación desde un tipo primitivo. También es posible el paso inverso, es decir obtener un tipo primitivo desde un objeto de envoltura.

```
Integer a = 25;
int b = 7;
a--;
Integer c = a + b;
```

Como se puede observar en el ejemplo anterior, los operadores aritméticos y lógicos también se han sobrecargado para que actúen sobre las clases de envoltura de la misma forma, para añadir nuevas funcionalidades, en que lo hacían sobre los tipos primitivos. Hay que tener cuidado especialmente con el operador de igualdad (==) ya que al usarlo sobre objetos de envoltura hay que recordar que son objetos y que pueden compartir el mismo valor aparente, pero no la misma dirección de memoria.

```
Integer a = 275;
Integer b = 275;
boolean c = (a == b); //Devulve falso
```

De igual forma las propiedades de *autoboxing* y *autounboxing* facilitan el uso de contenedores con tipos primitivos. Debe observarse que al declarar el contenedor se debe utilizar la clase envoltorio correspondiente en vez el tipo primitivo directamente, pues se precisa un tipo no primitivo. Sin embargo, luego se pueden utilizar las funciones de inserción y recuperación directamente sobre el tipo primitivo, pues el *autoboxing* y el *autounboxing* se encargan de las conversiones.

```
VectorDinamico <Integer> v = new VectorDinamico <Integer>();
v.poner(5);
int x = v.obtener();
```

4.8. Lecturas recomendadas

"*Effective Java*", 2ª edición, Joshua Bloch, Addison-Wesley, 2008. Gran libro para profundizar en la programación en *Java* que todo entendido debe leer de principio a fin. El libro se divide en pequeños capítulos que tratan aspectos específicos del lenguaje y de su correcto uso. El capítulo 3 trata la implementación de algunos de los métodos de `object` comentados aquí.

4.9. Ejercicios

Ejercicio 1

Se desea crear la clase `Lista` que implementa una lista de objetos de la clase `Object` y que implementa de manera pública al menos los métodos que se detallan a continuación.

```
void pushBack(Object o)         // Inserta por el final
void pushFront(Object o)        // Inserta por el principio
void popBack()                  // Elimina por el final
void popFront()                 // Elimina por el principio
Object front()                  // Devuelve el valor que está en el principio
Object back()                   // Devuelve el valor que está en el final
int size()                      // Devuelve el número de elementos de la lista
void clear()                    // Borra todos los elementos de la lista
```

a) Escribir en *Java* el código correspondiente a la clase `Lista` que implementa una lista de objetos de la clase `Object`. Esta clase debe contener al menos los métodos que se detallan a continuación.

b) Construir una clase `ListaEnteros1` que cuente con métodos análogos a los de la clase Lista pero que añada otros que sólo permitan almacenar enteros (objetos de la clase envoltorio `Integer`). Para ello usar herencia, es decir, hacer que `ListaEnteros1` herede de `Lista`.

c) Construir una clase `ListaEnteros2` que cuente con métodos análogos a los de la clase `Lista` pero que sólo permita almacenar enteros (objetos de la clase envoltorio `Integer`). Para ello usar composición, es decir hacer que `ListaEnteros2` tenga dentro un objeto de la clase `Lista`.

d) Decidir qué enfoque es mejor para construir la lista de enteros (tipo `ListaEnteros1` o tipo `ListaEnteros2`) y por qué razón. Explicar también por qué se debe usar la clase envoltorio `Integer` en vez de usar directamente el tipo primitivo `int` al almacenar los enteros en la lista.

Capítulo 5 Diseño de Clases

Básicamente, la construcción de programas orientados a objetos consiste en la definición de clases. En este capítulo se expondrán diferentes herramientas utilizables en un enfoque *top-down* apoyado en las diferentes herramientas que *UML* proporciona.

5.1. Diseño *top-down* apoyado en UML

Ya se ha comentado que la creación de clases aporta numerosas ventajas: es un buen mecanismo para encapsular y reutilizar el código, acerca el vocabulario del lenguaje de programación al del problema a resolver, etc.

En las etapas iniciales de desarrollo de un proyecto se suelen usar algunos métodos para descubrir las clases que se deben crear. Uno de ellos consiste en identificar las clases con los sustantivos de una descripción del problema. Otro consiste en escribir tarjetas que representan las clases y acompañarlas de una descripción de sus responsabilidades. En cualquier caso, estos primeros pasos permiten obtener un conjunto de clases que luego se va completando con nuevas clases siguiendo un enfoque *top-down*.

Desde un punto de vista puramente constructivo, la creación de una clase nueva se realiza utilizando tres tipos de relaciones:

- **Las relaciones de dependencia entre objetos.** Creando una clase que utiliza a otras como parte de sus métodos (en la implementación o en los parámetros que recibe).
- **Las relaciones de asociación entre objetos**. Ensamblando diferentes objetos se construye un objeto mayor que los contiene.
- **Las relaciones entre clases e interfaces.** Utilizando herencia se construyen jerarquías de abstracciones que se basan unas en otras.

En esta sección se estudiará en detalle cada uno de estos tipos de relaciones. Su estudio se hará desde dos perspectivas: una estática y otra dinámica. La estática mostrará cómo se estructuran las clases y los objetos. La dinámica cómo interactúan los objetos y las clases para resolver diferentes situaciones.

Además, estos tres tipos de relaciones se revisarán respecto a tres aspectos: la **versatilidad**, la **temporalidad** y la **visibilidad**. El estudio de estos aspectos, en cada problema particular, puede ayudar a elegir el tipo de relación que debe usarse al crear nuevas clases.

Versatilidad

Un objeto se dice que es versátil si puede combinarse con otros objetos de diversas maneras para dar lugar a diferentes comportamientos. Por ejemplo, una clase `Pila` que pueda almacenar cualquier tipo de objetos es una clase más versátil que una clase `Pila` que sólo permita almacenar números enteros.

En general, es preferible hacer código versátil, pues facilita su reutilización en diferentes problemas. Sin embargo, un código excesivamente versátil puede ser que se ajuste poco a un problema particular que se esté resolviendo y que, con ello, se dificulte la programación en vez de simplificarla. Por tanto, al desarrollar un programa, es importante estudiar cuál es el nivel de versatilidad que interesa para las clases e interfaces que se van a crear, en el contexto del problema que se está resolviendo.

Temporalidad

Cualquier relación entre objetos o entre clases tiene una duración temporal. Hay relaciones entre objetos que se dan en un ámbito muy restringido (por ejemplo dentro de un método) y hay relaciones entre objetos que abarcan toda la vida de los objetos (por ejemplo en las propiedades). Veremos que debe prestarse atención al estudio de la temporalidad de una relación porque ayuda a definir el tipo de relación, aunque en igualdad de condiciones se prefiere una baja temporalidad, pues esto implica mayor independencia y menor coste de recursos.

Visibilidad

Con objeto de maximizar el nivel de encapsulación, la visibilidad que una clase tiene de otra es otro aspecto que debe tenerse en cuenta al crear relaciones. La visibilidad se restringe definiendo cuáles son los miembros públicos, privados y protegidos, y cuáles son las interfaces y clases públicas.

Esta claro que en una relación entre dos clases u objetos siempre es preciso que exista algún nivel de visibilidad para que puedan realizarse interacciones. En general, se debe seguir el criterio de definir la mínima visibilidad posible que permita obtener la funcionalidad requerida. De esta forma se reduce la complejidad manteniendo la funcionalidad.

Otros criterios

Existen otros aspectos, como la economía y la eficiencia, que pueden condicionar el diseño de una relación. El utilizar estos criterios puede chocar frontalmente con un buen diseño orientado a objetos. Por ejemplo, se puede imaginar un caso en el que tras detectar un problema se proponen dos soluciones. La primera consiste en rediseñar una clase. La segunda en permitir el acceso a cierta variable desde fuera de la clase. La primera solución podría ser óptima en cuanto a cumplir el contrato del objeto, mientras que la segunda podría crear consecuencias inesperadas si los objetos clientes utilizan la variable sin cuidado. Sin embargo la primera implica varias horas de desarrollo y la segunda sólo unos segundos. Si los plazos son ajustados seguramente se optará por la segunda opción. Es decir, en los proyectos comerciales, la existencia de plazos definidos sujetos a penalizaciones económicas suele condicionar detalles de la implementación.

5.2. Relaciones de dependencia entre objetos.

La relación de dependencia se da cuando una clase depende de la interfaz de otra. Las dependencias entre clases existen por dos motivos:

- Un método de una clase recibe como parámetro un objeto de otra clase
- Una clase crea objetos de otra dentro de un método.

Dimensión estática

En *UML* la representación estática de las relaciones de dependencia entre dos clases se realiza utilizando una flecha discontinua. Esta flecha parte de la caja que representa a la clase de los objetos que usan, y termina en la clase de los objetos usados.

Figura 26.- La clase `A` depende de la clase `B`.

Dimensión dinámica

La dimensión dinámica se plasma en *UML* mediante los **Diagramas de Interacción** entre objetos (ver Figura 27). Estos diagramas describen, utilizando flechas y cajas, cómo interactúan los objetos de una relación de dependencia.

UML define dos tipos de Diagramas de Interacción: los **Diagramas de Secuencia** y los **Diagramas de Colaboración**. La Figura 27 muestra un Diagrama de Secuencia y la Figura 28 muestra uno de colaboración. En ambos diagramas se describe la siguiente situación: un objeto de la clase Trafico, identificado como `t`, invoca al método frenar de un objeto de la clase Coche, identificado como `c`. Dentro de ese método `frenar()`, se invoca al método `reducirEntradaGasolina()` de un objeto de la clase Motor referenciado como `m`. Luego, aún dentro del método frenar de `Coche`, se invoca, sobre el propio `c` al método `frenarRuedas()`. Finalmente, el control retorna al invocador de los métodos.

Como se puede apreciar ambos tipos de diagramas son equivalentes. Sin embargo, en el Diagrama de Secuencia, se ve mejor el orden de llamada a los métodos. Mientras que el Diagrama de Colaboración es más compacto y quizás más cómodo para realizar la primera aproximación a un problema.

Los Diagramas de Interacción plantean escenarios sobre los que se descubren los métodos de las clases, su visibilidad y las relaciones existentes entre clases y objetos. Al plantear un Diagrama de Interacción se le suele poner un título descriptivo (por ejemplo: "secuencia de frenado"). El diagrama se limita a describir las llamadas que se producen entre los objetos para conseguir el objetivo enunciado en dicho título.

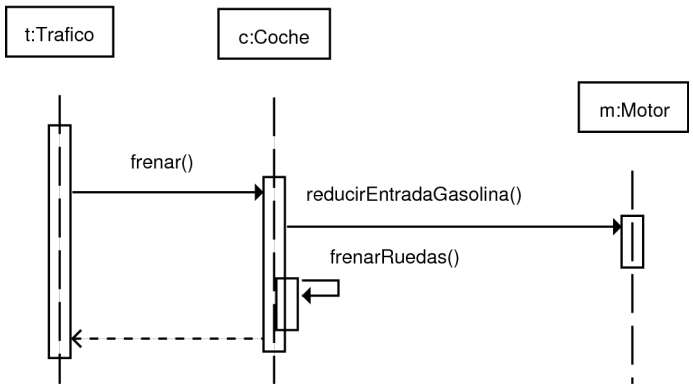


Figura 27.- Diagrama de Secuencia.

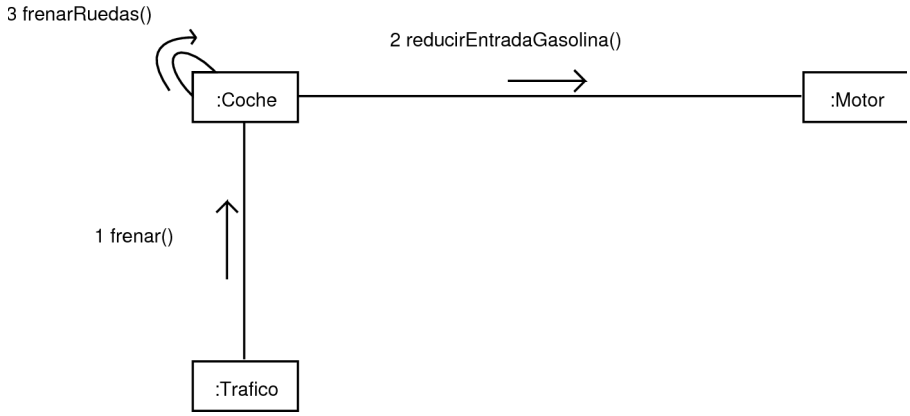


Figura 28.- Diagrama de Colaboración.

Sobre los Diagramas de Secuencia se pueden especificar varios detalles adicionales. En la Figura 29 se presenta la creación de un objeto desde otro y su posterior destrucción, en este caso por dejar de estar referenciado. En la Figura 30 se muestra la devolución de un parámetro y el uso de una instrucción condicional, obsérvese que la condición se escribe entre corchetes. Finalmente, la Figura 31 muestra un caso que incluye un bucle.

A pesar de la riqueza de estos detalles, no debe pretenderse que los Diagramas de Secuencia sean completos. Es decir, cualquier omisión en un Diagrama de Secuencia solo indica que dicho detalle no se considera importante, no que deba omitirse en la implementación. Siempre debe recordarse que los Diagramas UML no son un fin, sino una herramienta para entender los problemas.

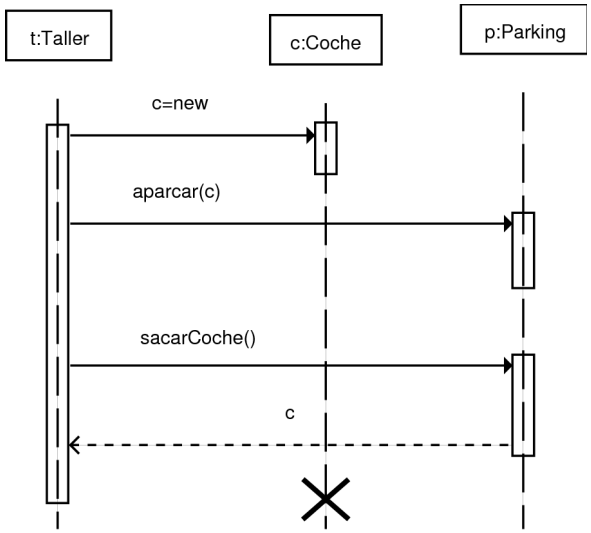


Figura 29.- Ejemplo de Diagrama de Secuencia con creación y finalización de objetos.

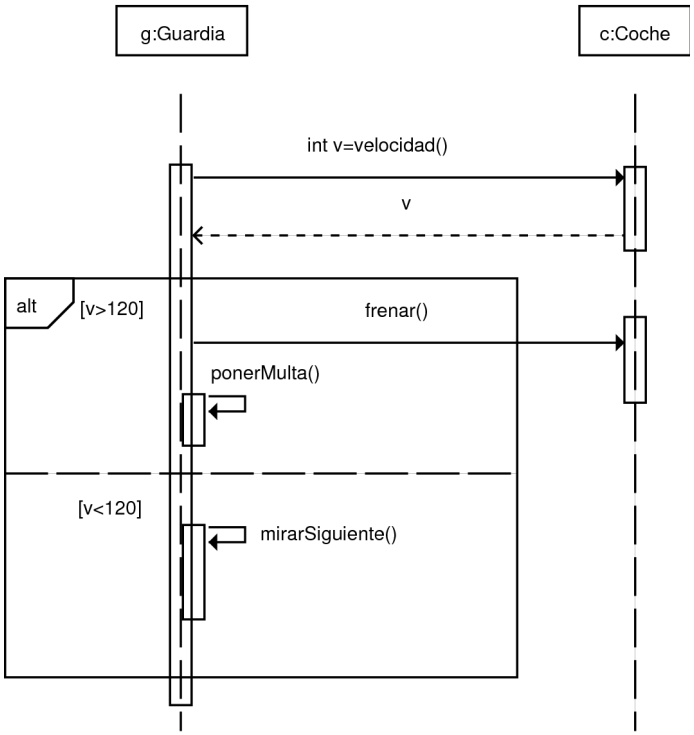


Figura 30.- Ejemplo de Diagrama de Secuencia con retorno de valores y con instrucción condicional.

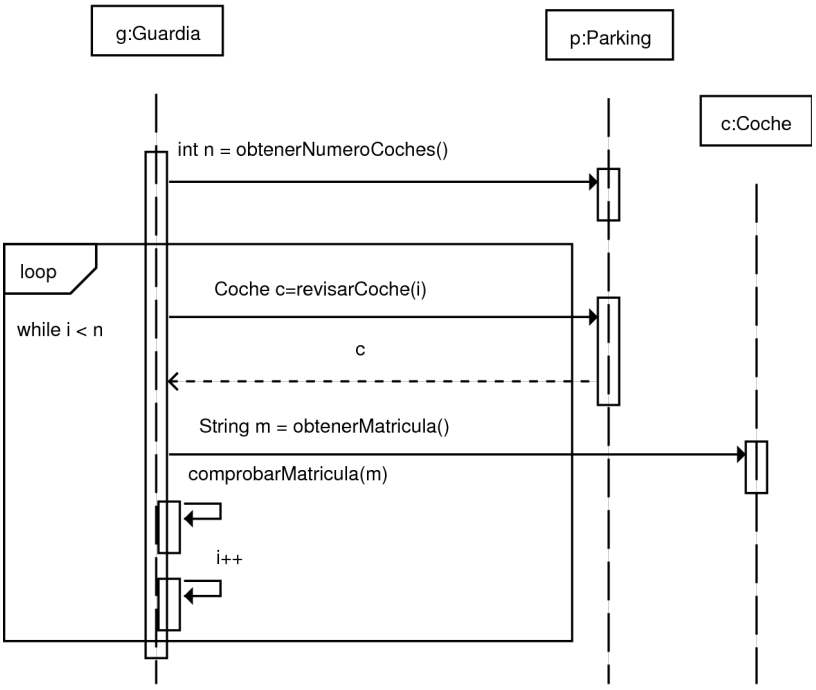


Figura 31.- Ejemplo de Diagrama de Secuencia con un bucle.

Cuándo y cómo usar relaciones de dependencia

Debe utilizarse cuando la temporalidad de una relación sea baja. Si la relación entre dos objetos se limita al ámbito de un método, la relación de dependencia es ideal. Si la relación entre dos objetos es más larga, debe buscarse una relación con mayor temporalidad, como las relaciones de asociación.

En la relaciones de dependencia debe fomentarse la versatilidad, utilizando el polimorfismo en el paso de parámetros. Cuanto más básicos sean los tipos de los parámetros de un método, más versátil será el objeto.

Desde el punto de vista de la visibilidad las relaciones de dependencia favorecen la encapsulación de los participantes en la relación, ya que los objetos sólo se conocen a través de su interfaz pública.

Ejemplo de relaciones de dependencia

Es habitual que el programador sin experiencia en Programación Orientada a Objetos tienda a construir objetos grandes, con multitud de métodos que le permiten desempeñar cierta tarea. Estos objetos, que podríamos denominar monolíticos, son complejos y en general menos versátiles que si se hubiesen construido múltiples objetos que se utilizasen entre ellos.

Si deseamos construir una aplicación para formatear textos de manera que permita eliminar espacios consecutivos innecesarios, poner en mayúsculas la primera letra tras un punto, obtener el número de palabras del texto, eliminar palabras repetidas resultado de un error al escribir, etc. Podríamos hacerlo de dos formas:

- Mediante una clase `Formateador` que dispusiese de un método formatear texto y de los siguientes métodos privados: `eliminarEspacios`, `capitalizar`, `eliminarRepeticiones`, `obtenerNumeroPalabras`.
- Creando una clase diferente por cada operación que se desea crear (`EliminadorDeEspacios`, `Capitalizador`, `EliminadorDeRepeticiones`, `ContadorDePalabras`) y luego creando una clase `Formateador` que utiliza esas clases para formatear un texto.

Si adoptamos la primera solución será difícil que varias personas aborden el desarrollo simultáneamente, debido a que todo el código estará en un único fichero. Además, dicho fichero será grande y complejo. Mientras que la segunda solución permite que diferentes programadores aborden el desarrollo de cada una de las diferentes clases, las cuales además serán relativamente simples.

Por otro lado, si posteriormente se desea utilizar una de las características por separado (por ejemplo la de eliminación de palabras repetidas), en el caso de la primera solución deberemos hacer público el método correspondiente o complicar la configuración del objeto `Formateador` para que permita elegir qué operaciones se aplican en cada caso. Por contra, la segunda solución no exige ningún cambio.

Quizás el programador novel piense que la segunda solución tenderá a sobrecargar el sistema debido a la multitud de objetos involucrados en cada formateo de texto, pero siempre se debe recordar que el coste de creación de un objeto es despreciable respecto a la versatilidad que aporta el enfoque.

Se puede concluir que la segunda solución es preferible por simplicidad de desarrollo, porque proporciona una colección de objetos mucho más versátil y fácil de mantener que la primera solución.

5.3. Relaciones de asociación entre objetos

Las relaciones de asociación surgen cuando se unen varios objetos para formar uno nuevo. La interacción entre los objetos que se unen hace que emerja un comportamiento mayor que la simple suma de los comportamientos de sus elementos.

Aunque una asociación entre objetos puede ser algo meramente conceptual, en muchas ocasiones la asociación se materializa en propiedades de clase. En estos casos, como consecuencia de que un objeto `a` de la clase `A` esté asociado a un objeto `b` de la clase `B`, la clase `A` tendrá una propiedad que permite referenciar a objetos de la clase `B`.

Una asociación entre objetos se puede estudiar desde dos perspectivas diferentes: una perspectiva estructural (estática) y otra de comportamiento (dinámica).

Dimensión estática

La dimensión estática debe mostrar cómo se relacionan las clases y las interfaces de los objetos que se asocian. Ya vimos en el capítulo 1 que los Diagramas Estáticos de Clases muestran la asociación entre dos objetos de dos formas:

- Mediante una línea que une las cajas que representan a cada clase de objetos.
- Mediante una propiedad.

Se usa una línea cuando se desea recalcar la importancia de la asociación y una propiedad cuando se considera menos importante. Por ejemplo, imaginemos un programa de contabilidad en el que para almacenar los empleados de una empresa los objetos de una clase llamada `Empresa` tienen una asociación con los objetos de una clase llamada `Persona`. Supongamos que para guardar el nombre de las personas también sea necesaria una asociación entre los objetos de la clase `Persona` y los de la clase `String`. En el primer caso podría ser interesante representar las clases `Empresa` y `Persona` y una línea entre ellas para remarcar la asociación. Por otro lado, la asociación entre `Persona` y `String` no parece tan importante y podría expresarse mediante una propiedad.

Figura 32.- Algunas asociaciones entre objetos se plasman en *UML* con una línea de asociación y otras, menos importantes, aparecen como propiedades.

En las líneas de asociación se pueden añadir una serie de adornos para aumentar el detalle de la descripción de la asociación.

- Por ejemplo, como vimos en el capítulo 1, se pueden indicar las cardinalidades de los objetos que se asocian.
- Además, en ambos extremos de la línea pueden ponerse identificadores que se correspondan con las propiedades que en el código implementarán la asociación. Estos identificadores pueden ir precedidos del símbolo negativo, positivo o de la almohadilla para indicar respectivamente que son privadas, públicas o protegidas.
- También puede añadirse un identificador que da nombre a la asociación y que lleva asociado un marcador en forma de flecha que indica el sentido de lectura.
- Finalmente, se pueden añadir terminaciones en flecha en ambos extremos de la línea de asociación para indicar que desde la clase origen de la flecha existe un acceso a los objetos de la clase destino de la flecha.

El diagrama de la Figura 33 muestra una asociación entre las clases `A` y `B`. De él se desprende que un objeto de la clase `A` se asocia a un objeto de la clase `B` utilizando una propiedad cuyo identificador es privado y se llama `objetoB`. También puede leerse en este diagrama que un objeto de la clase `B` se asocia a un único objeto de la clase `A`. Además, la flecha indica que en la clase `A` debe de haber métodos para acceder a la propiedad `objetoB`, y el aspa indica que en la clase `B` no los hay para acceder a la `A`.

Por ejemplo, si se tiene la clase `Coche` y la clase `Rueda`, se puede establecer una relación de asociación entre ambas que diga: un coche tiene 4 ruedas. Así, la Figura 34 muestra que un coche tiene cuatro ruedas y que una rueda puede pertenecer a un sólo coche o a ninguno. Además, la clase coche tiene una propiedad privada, llamada `rueda`, que almacena 4 referencias a objetos de la clase `Rueda`.

Figura 33.- Relación de asociación entre las clases `A` y `B` en la que `B` es navegable desde `A`, pero no viceversa.

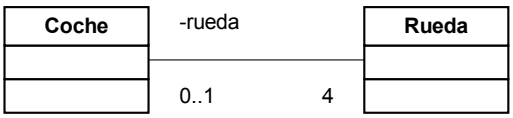


Figura 34.- Asociación entre la clase `Coche` y la clase `Rueda`.

En ocasiones es útil distinguir entre dos tipos de relaciones de asociación:

- Relaciones de Agregación.
- Relaciones de Composición.

Relaciones de Agregación.- Son asociaciones en las que una parte contiene a elementos de otra parte. Normalmente responden a la pregunta "tiene un". El diagrama de la Figura 35 presenta una agregación en la que un objeto de la clase `A` puede contener otros de la clase `B`. Según *UML* la relación de agregación solo añade información semántica, aunque suele significar que el objeto contenedor tiene una referencia al objeto parte. Por ejemplo, la relación -un `Coche` tiene 4 `Ruedas`- es una relación de agregación.

Figura 35.- Relación de agregación. Los objetos de la clase `A` tiene referencias a objetos de la clase `B`.

Relaciones de Composición.- Son relaciones de agregación con una relación de pertenencia fuerte. Esta pertenencia suele implicar que el objeto contenido se implementa como una propiedad del objeto contenedor. Como consecuencia, tanto contenedor como contenido, una vez creados permanecen juntos hasta su destrucción.

Por ejemplo, si se tiene la clase `Empresa` y la clase `Departamentos`, la relación -una empresa "tiene varios" departamentos- es una relación de composición. Claramente la eliminación de un objeto de tipo `Empresa` implica eliminar los objetos de la clase `Departamento`.

Figura 36.- Relación de composición. Los objetos de la clase `A` son propietarios de objetos de la clase `B`.

Por otro lado, en el ejemplo de agregación anterior existe una relación de pertenencia más débil entre los objetos `Rueda` y los objetos `Coche`. La existencia de un objeto `Coche` es independiente de la existencia de objetos `Rueda`. Si se destruye un objeto `Coche` el objeto `Rueda` puede seguir existiendo, por ejemplo para utilizarse en otro objeto `Coche`.

Finalmente se debe decir que la ausencia de estos adornos en un diagrama no implica que luego en el código no se implementen los elementos que no se detallan. Su ausencia sólo significa que no se ha considerado importante su inclusión en ese diagrama.

Dimensión dinámica

Al igual que en el caso de las relaciones de dependencia, la dimensión dinámica de las relaciones de asociación se representan en *UML* dentro de los Diagramas de Interacción.

Cuándo y cómo usar las relaciones de asociación

Las relaciones de asociación deben usarse cuando se prevé una temporalidad alta en la relación de dos objetos.

Respecto a la visibilidad debe fomentarse el uso de propiedades privadas. En caso necesario deberán definirse métodos con mayor visibilidad para consultar o cambiar el valor de tales propiedades.

Por último se debe potenciar la versatilidad utilizando el polimorfismo. Para ello, en la definición de las propiedades, debe preferirse utilizar referencias a clases bases frente a clases derivadas.

Ejemplo del programa de facturación

Para ilustrar los conceptos de asociación y dependencia recién expuestos se propone diseñar un programa que permita crear e imprimir las facturas de una tienda de venta de recambios de automóviles. El programa deberá permitir crear facturas e imprimirlas. Las facturas contendrán la información relativa al cliente (nombre e identificación fiscal) y a las piezas que compra (nombre y precio).

Primeramente se realiza el Diagrama Estático de Clases de la Figura 37. Este diagrama incluye varios elementos que se deducen directamente del enunciado. Así, incluye la clase `FacturacionRecambios` que contiene el método `main`, donde comienza la ejecución del programa. También se deducen del enunciado las clases `Pieza` y `Cliente`. Estas clases se encargan respectivamente de recoger y mantener la información relativa a las piezas y al cliente. La clase `Factura`, que se encarga de contener piezas y clientes y de imprimir el resultado, también se deduce del enunciado. Finalmente, la clase `Menu` se encarga de permitir navegar entre las opciones de crear facturas e imprimir.

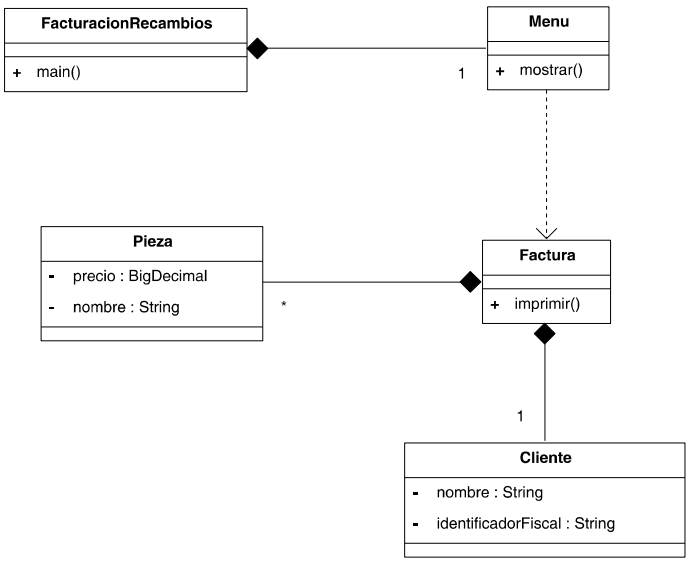


Figura 37.- Diagrama de clases inicial para el programa de facturación.

Tras este diagrama de clases preliminar, se procede a crear Diagramas de Secuencia basados en diferentes escenarios. Así se crean diagramas para los escenarios de: creación de una factura desde el menú (Figura 38), rellenado de los datos de una factura (Figura 39) e impresión de la misma (Figura 40).

Para ilustrar el carácter iterativo del proceso de descubrimiento de clases y métodos, finalmente, en la Figura 41, se presenta un diagrama de clases que recoge los métodos descubiertos al crear los Diagramas de Secuencia. Este proceso iterativo de descubrimiento y refinamiento es la base del diseño basado en el análisis del problema.

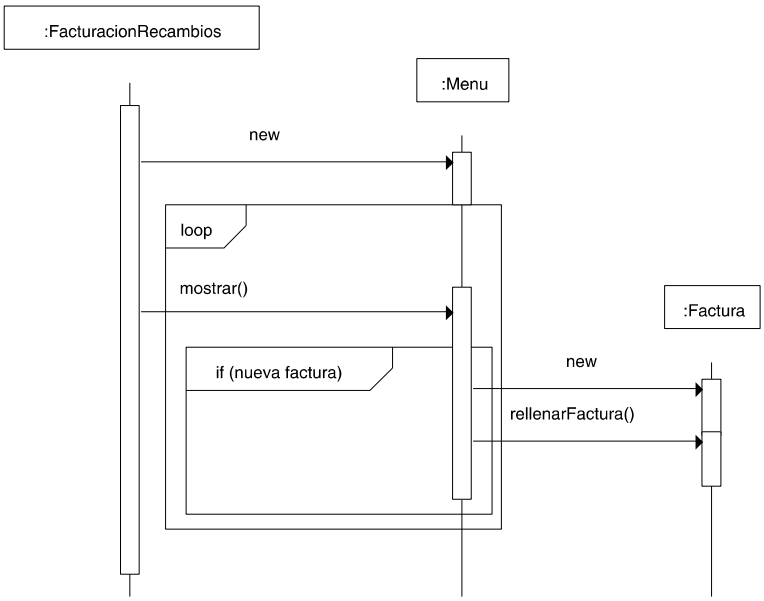


Figura 38.- Diagrama de secuencia que refleja la creación de una nueva factura desde el menú.

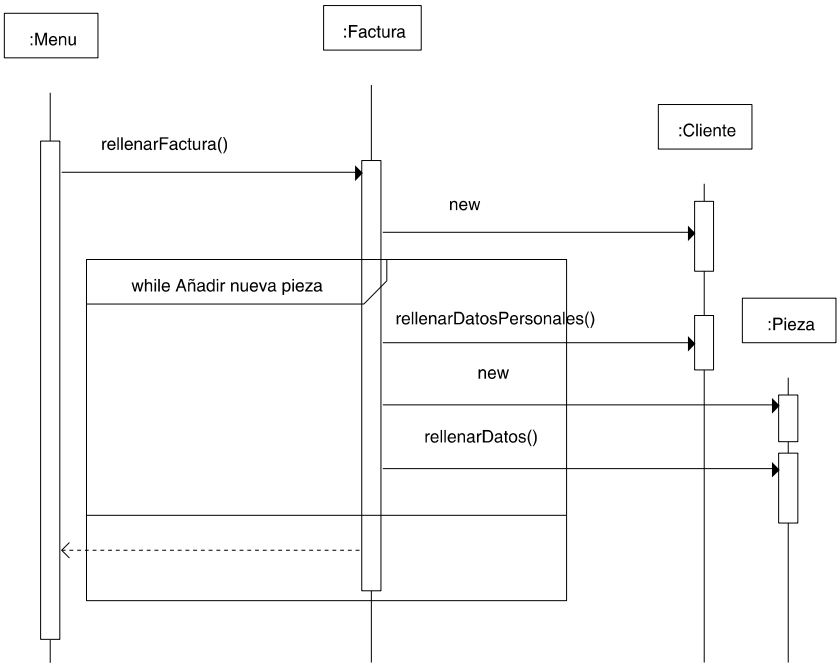


Figura 39.- Diagrama de secuencia que refleja cómo se rellenan los datos de una factura.

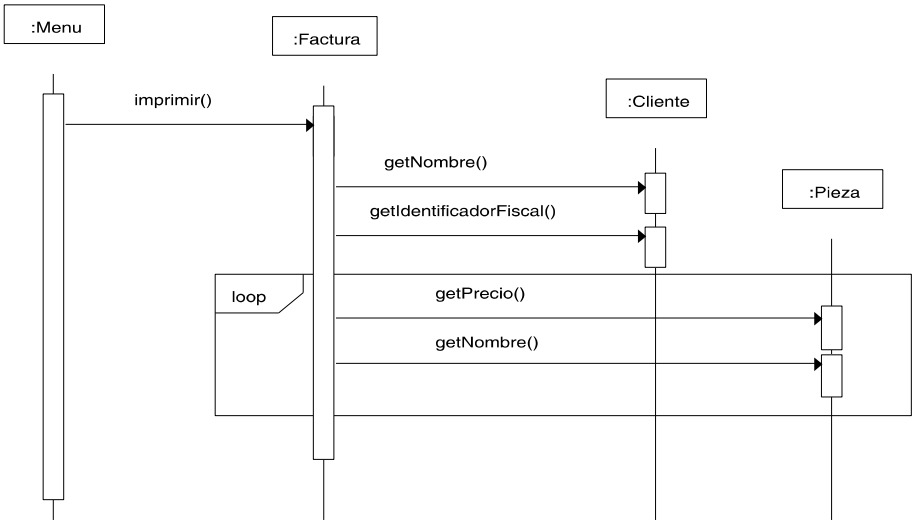


Figura 40.- Diagrama de secuencia que refleja la impresión de una factura desde el menú.

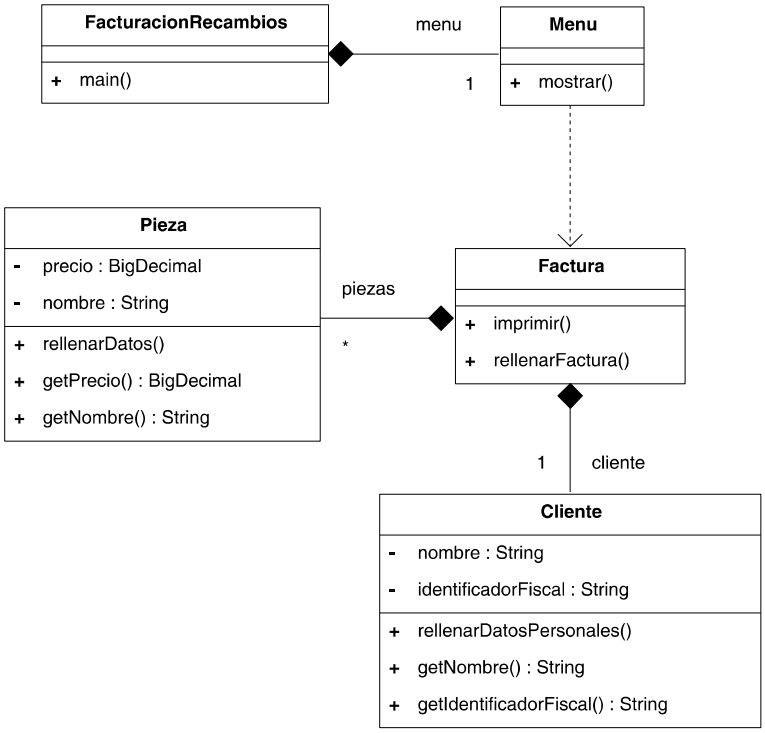


Figura 41.- Diagrama de clases con los métodos encontrados en los diagramas de secuencia.

5.4. Relaciones entre clases

Ya hemos visto que la herencia es un mecanismo, proporcionado por los lenguajes orientados a objetos, que permite crear jerarquías entre clases e interfaces. Respecto a la herencia de implementación, las jerarquías deben buscar las similitudes que deben implementarse en las clases bases para que las clases derivadas las aprovechen. Respecto a la herencia de interfaces, las jerarquías deben definirse de manera que las interfaces derivadas se consideren casos concretos de las interfaces bases, facilitando los comportamientos polimórficos.

Cuándo utilizar el polimorfismo

Al hablar de cómo se deben construir las jerarquías de clases resulta imprescindible tener en cuenta el polimorfismo.

Por un lado, el polimorfismo estático, es decir, el uso de varios métodos con el mismo nombre pero con diferentes parámetros dentro de una clase, debe utilizarse para proporcionar uniformidad a aquellos métodos que conceptualmente hacen lo mismo pero que lo hacen partiendo de diferentes elementos.

Por otro lado, al tratar con jerarquías de clases, a menudo se desea tratar un objeto de una forma genérica e independiente del tipo específico al que pertenece. El polimorfismo dinámico se utiliza en este caso para usar una misma referencia sobre distintos objetos de una jerarquía de clases. Esto permite realizar operaciones genéricas sobre diferentes clases de objetos (que comparten una misma interfaz) sin tener que estar pendiente en cada momento de qué tipo de objeto está siendo tratado.

Así, cuando se necesite que un método de una clase se comporte de una forma u otra dependiendo de una propiedad de la clase, puede ser mejor heredar y crear diferentes clases con diferentes implementaciones para tal método. Esto haría que si apareciesen nuevos comportamientos no fuese necesario tocar el código ya existente, sino añadir nuevas clases. De esta forma, el polimorfismo dinámico fomenta la reutilización de código sin romper la encapsulación.

Crear clases abstractas o crear interfaces

Ya se ha dicho que una clase abstracta es una clase de la que no se pueden crear instancias de objetos debido a la inexistencia de implementación para alguno de sus métodos. Crear clases abstractas simplifica las interacciones con conjuntos de objetos que comparten cierto tipo porque abstraen interfaz y comportamiento. Por ejemplo, si de la clase abstracta `Coche` heredan varias clases (`Ford`, `Renault`, `Seat`, `Citröen`), se podría definir la clase `Garaje` que interactúa con objetos que cumplan el tipo definido por la clase `Coche` sin conocer exactamente de qué clase particular son estos objetos.

También se ha dicho que en *Java* sólo existe herencia simple entre clases, es decir, una clase no puede heredar de más de una clase simultáneamente. Al diseñar *Java* se tomó esta decisión para eliminar el problema de la ambigüedad (ver 1.2.3). Así, los diseñadores de *Java*, en su afán por simplificar aquellos aspectos de otros lenguajes (como *C++*) que complicaban la programación, prohibieron la herencia múltiple de clases. Sin embargo, esta restricción afecta de manera importante al polimorfismo dinámico, al impedir que un mismo objeto se comporte como un elemento genérico de dos o más jerarquías de clases diferentes.

Por otro lado, si las clases de las que se hereda no tuviesen implementado ningún método, el problema de ambigüedad no existiría. Por eso, en *Java* se permitió que una clase pudiese heredar de varias interfaces simultáneamente, ya que esto no implica ninguna ambigüedad, y elimina los efectos negativos sobre el polimorfismo derivados de la prohibición de la herencia múltiple de clases. De nuevo, usar adecuadamente la herencia de interfaz abstrae las interacciones, al permitir que un mismo objeto cumpla varios tipos simultáneamente.

En general, suelen utilizarse clases abstractas cuando además de una interfaz genérica se desea añadir cierto comportamiento a todos los elementos de una jerarquía. Por otro lado, suelen utilizarse interfaces cuando sólo se desea asegurar que un conjunto de objetos cumple cierta característica que lo hace tratable por otro conjunto de objetos.

Dimensión estática

Ya se ha explicado que en *UML* la herencia se representa, en el Diagrama Estático de Clases, mediante una flecha de punta hueca que parte de la clase que hereda y termina en la clase padre, y que esta flecha es punteada si la herencia es de interfaz. En la Figura 42 tenemos tres interfaces y dos clases. La clase `Quad` hereda de `VehiculoAMotor,` pero además implementa las interfaces definidas por `Automovil` y `Motocicleta,` las cuales a su vez derivan de `Vehiculo`. Además, el ejemplo ilustra la inexistencia del problema de ambigüedad cuando solo existe herencia simple de clases (`Quad` hereda de una sola clase) aunque exista herencia múltiple de interfaz (`Quad` implementa varias interfaces). Obsérvese que, cuando un método es abstracto se representa en letras cursivas, por lo que el método `frenar()` solo está implementado en la clase `Quad.` De la misma forma, el método `acelerar()` solo puede heredarlo `Quad` de `VehiculoAMotor.`

Dimensión dinámica

Un **Diagrama de Actividad** es fundamentalmente un diagrama de flujo que muestra la secuencia de control entre diferentes acciones, mostrando tanto la concurrencia como las bifurcaciones de control. Los Diagramas de Actividades pueden servir para visualizar, especificar, construir y documentar desde la dinámica de una sociedad de objetos, hasta el flujo de control de una única operación. La ejecución de una actividad se descompone en la ejecución de varias acciones individuales cada una de las cuales pueden cambiar el estado de un objeto o pueden comunicar mensajes a otros objetos.

Los elementos básicos de un Diagrama de Actividad son:

- Las acciones y nodos de actividad
- Objetos de valor
- Flujos de control y de objetos

Los sucesos que describe un Diagrama de Actividad ocurren en las acciones y en los nodos de actividad. Cuando el suceso es atómico, no puede descomponerse, es una acción. Mientras que, cuando el suceso puede descomponerse en sucesos más elementales es un nodo de actividad o actividad.

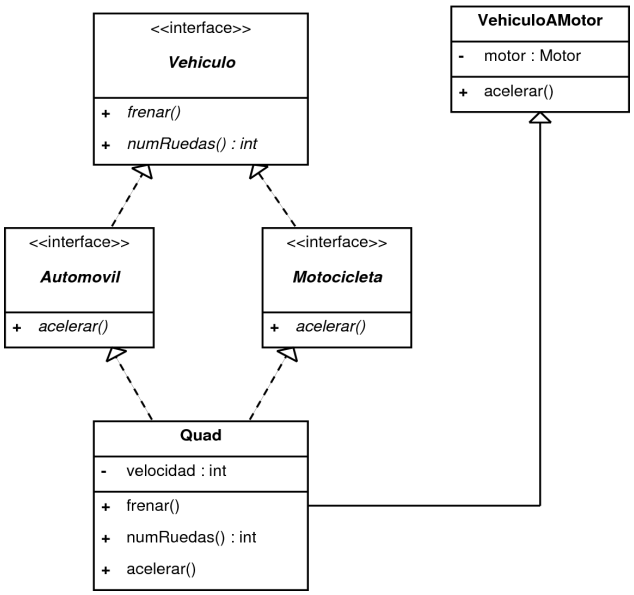


Figura 42.- Ejemplo de herencia interfaces. Obsérvese que no hay problema de ambigüedad porque las interfaces no implementan ninguno de los métodos que declaran.

Tanto las acciones como los nodos de actividad se representan mediante un rectángulo con las esquinas redondeadas (véase la Figura 43) y un texto descriptivo en su interior. En el caso de las acciones el texto se corresponde a una operación simple o a una expresión, mientras que en los nodos de actividad el texto se corresponde al nombre de otra actividad.

En general, una acción puede verse como un caso particular de una actividad. En adelante, por simplicidad se hablará exclusivamente de actividades, aunque todo sería igualmente aplicable a acciones.

En los Diagramas de Actividad se pueden representar instancias de los objetos involucrados mediante rectángulos (véase la Figura 43). En el interior de estos rectángulos se describe mediante texto la naturaleza del objeto y su valor. Además se puede etiquetar, mediante un texto entrecomillado, que se denomina estereotipo, categorías especialmente importantes de objetos (como por ejemplo los almacenes).

Los objetos de un Diagrama de Actividad pueden corresponder a elementos producidos, consumidos o utilizados por una actividad.

Los flujos de control dentro de un Diagrama de Actividad marcan el orden en el que se pasa de una actividad a otra. Los flujos de control se detallan mediante flechas que unen las cajas de las actividades (véase la Figura 44).

Es posible especificar el principio de un flujo de control mediante un círculo relleno. Igualmente, el fin de un flujo de control se puede especificar mediante un círculo relleno concéntrico a una circunferencia exterior.

Dentro de los flujos de control es posible especificar bifurcaciones mediante rombos huecos. Al rombo llega una flecha de flujo de control y salen tantas flechas como bifurcaciones se produzcan. Acompañando a cada flecha de salida del rombo se escribe un texto entre corchetes, que se denomina guarda, con la condición que determina que se siga ese camino.

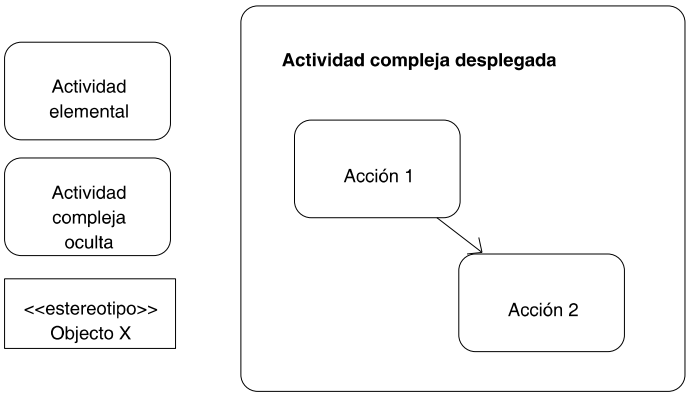


Figura 43.- Nodos de un Diagrama de Actividad. Los rectángulos redondeados corresponden a acciones o a actividades, mientras que los rectángulos normales corresponden a objetos. Obsérvese que se suelen permitir licencias como sombreados o colores.

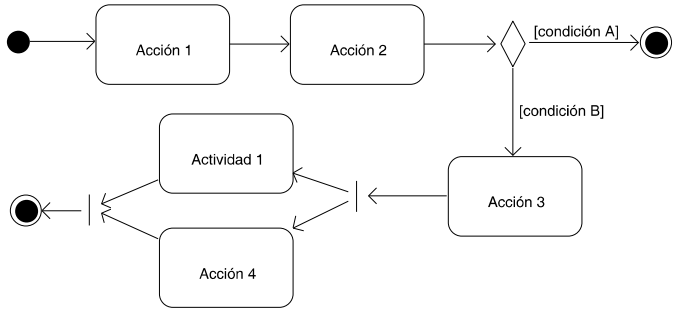


Figura 44.- Ejemplo de flujo de control. El punto relleno muestra el punto de inicio de flujo de control, los puntos concéntricos son puntos de fin del flujo, el rombo es una bifurcación y las barras verticales son puntos de sincronización donde se crean o se unifican hilos separados de ejecución.

También es posible especificar flujos concurrentes en los Diagramas de Actividades. Los puntos en los que el control se divide o se une en múltiples hilos se denominan barras de sincronización y se representan mediante un segmento horizontal o vertical de trazo grueso.

Cuando en un Diagrama de Actividad se involucran objetos es preciso detallar qué actividad los produce o consume. Así, cuando un objeto recibe una flecha de una actividad, significa que el objeto es producido por ella. Por otro lado, cuando existe una flecha de un objeto a una actividad significa que la actividad consume dicho objeto.

A partir de la versión 2.1 de UML, es posible añadir una pequeña caja, denominada pin, al contorno de una actividad para indicar de manera abreviada que esa actividad crea o consume un objeto. En este caso, sobre la flecha se escribe un texto que describe al objeto involucrado (véase la Figura 45).

Finalmente, es posible enlazar dos objetos mediante una flecha para indicar que los objetos interaccionan entre sí.

Los Diagramas de Actividad se pueden usar para detallar el comportamiento dinámico de un objeto cuando su vida transcurre a través de métodos implementados en diferentes clases, incluso cuando una parte está en una clase base y otra parte está en una clase derivada. Para ello, las operaciones se anidan dentro rectángulos llamados **calles**. Cada calle dispone de una etiqueta que indica cuál es el método y la clase en la que se encuentra esa parte de la implementación.

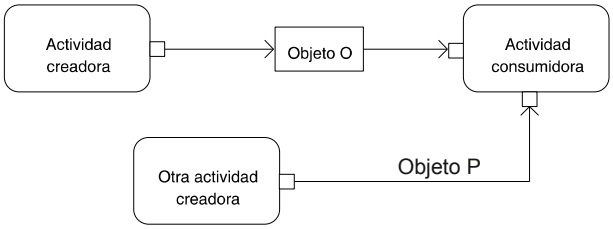


Figura 45.- Ejemplo de flujo de objetos. Arriba una "actividad creadora" crea un objeto y se lo envía al objeto O, éste a su vez genera un objeto que lo consume la "actividad consumidora". Abajo otra "actividad creadora" crea un objeto P que lo envía a la "actividad consumidora".

.

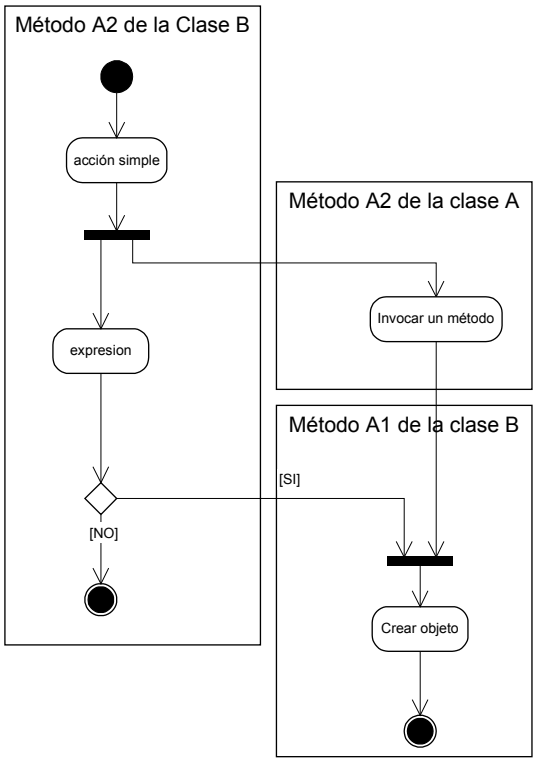


Figura 46.- Ejemplo de Diagrama de Actividad con calles.

Cuándo y cómo utilizar las relaciones de herencia.

Las relaciones de herencia tienen una alta temporalidad en comparación con las relaciones de asociación y de uso. Si una clase deriva de otra, lo seguirá haciendo mientras exista el programa.

En algunas ocasiones esta alta temporalidad es útil. Por ejemplo, si en el sistema educativo puede haber dos tipos de profesores (fijos e interinos), y para cada uno se calcula el sueldo de una forma diferente se puede usar herencia para implementar los dos tipos de profesores (ya que esta característica del sistema educativo no cambiará) y se particularizará la forma de cálculo del sueldo en cada clase derivada.

La herencia debe aportar versatilidad a los lenguajes de programación. En particular, el polimorfismo dinámico debe promover la versatilidad en los elementos que se desarrollan, mediante la definición de referencias comunes a objetos que comparten la misma jerarquía.

Desde el punto de vista de la visibilidad la herencia normalmente viola los principios de encapsulación. Esto se debe a que suele ser preciso conocer bastante a fondo la clase de la que se hereda. Para conseguir que esta visibilidad se restrinja sólo a las clases derivadas debe cuidarse la selección de los métodos protegidos. Además, debe limitarse en lo posible la visibilidad de las propiedades, fomentando el acceso a las mismas a través de métodos, de forma que se impidan un mal uso de la clase base. Por ello, en *Java*, se recomienda el uso del modificador `final`, tanto al declarar las clases, como en los métodos, pues con ello se impide la herencia. Solo en los casos en los que se piense que la herencia aporta cierta ventaja se debe omitir el modificador `final` y documentando claramente cómo debe realizarse dicha herencia.

Continuación del ejemplo de la aplicación de facturación

Se desea ampliar el programa de facturación de recambios con mecanismos de persistencia sobre una base de datos. Así, el programa deberá permitir la consulta de precios, la consulta de clientes, el almacenamiento de clientes y el almacenamiento de las facturas.

Con este objeto se crea una clase `ConsultaBD`. De esta clase, heredarán diversas clases que implementaran las diferentes operaciones a realizar contra la base de datos (ver Figura 47). Esta clase, que se utilizará desde la clase `Factura`, gestionará la conexión con la base de datos. Las clases derivadas se encargarán de implementar los detalles relativos a cada operación. Este enfoque tiene diferentes ventajas. Por un lado, permite aumentar el número de operaciones sin aumentar la complejidad de una sola clase. Por otro lado, permite centralizar operaciones comunes, como la conexión con la base de datos. El Diagrama de Actividad de la Figura 48 ilustra esta última posibilidad.

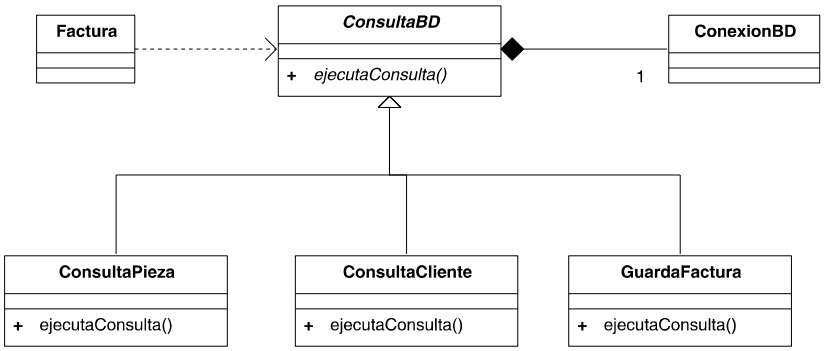


Figura 47.- Clases usadas para introducir persistencia en la aplicación de facturación.

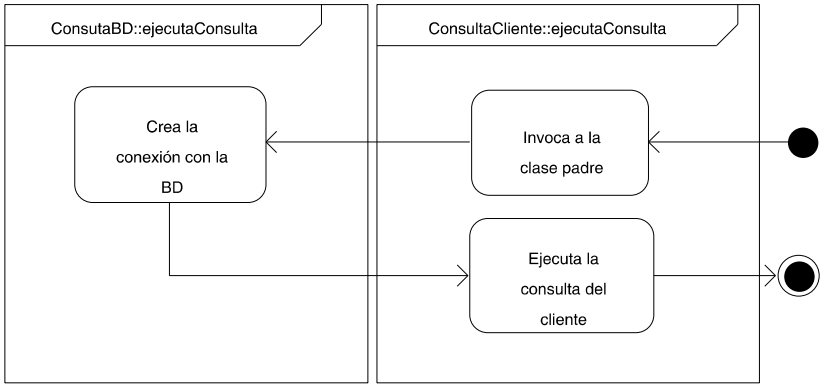


Figura 48.- Diagrama de Actividad que ilustra la división de tareas a través de la herencia en la persistencia del problema de facturación.

5.5. Conclusiones

Se ha visto que las relaciones de asociación ayudan a mantener la encapsulación, mientras que las relaciones de herencia tienden a violarla al precisar mucha visibilidad. Además, la herencia suele hacer crecer las interfaces de las clases dispersando su comportamiento, convirtiéndolas en elementos difíciles de gestionar y utilizar. Por ello, debe favorecerse la asociación de objetos frente a la herencia de clases cuando el objetivo es crear objetos con nuevas funcionalidades.

Por otro lado, la herencia debe utilizarse para crear interfaces comunes a conjuntos de clases que comparten cierta funcionalidad. En estos casos, las clases derivadas no hacen más que implementar o especializar el comportamiento de las clases padres.

Finalmente se debe señalar que la experiencia ha demostrado que el diseño no debe llegar a límites absurdos. Es decir, el diseño es una buena herramienta en las primeras etapas de desarrollo, para pensar sobre el desarrollo, para coordinar al equipo de trabajo y para estimar la cantidad de trabajo a realizar. También es una buena herramienta de documentación, pues permite entender más fácilmente la estructura de un programa que mediante la inspección del código o la lectura de descripciones textuales. Sin embargo, es innecesario, y aún contraproducente, que el diseño abarque todas las particularidades de cada clase y de cada método.

5.6. Lecturas recomendadas

"*UML* Gota a gota", Martin Fowler, Pearson,1999. Es un libro breve de imprescindible lectura para introducirse en el diseño orientado a objetos mediante UML.

5.7. Ejercicios

Ejercicio 1

¿Qué utilidad puede tener un Diagrama de Actividad? ¿Y uno de secuencia? ¿Son equivalentes los Diagramas de Actividad y de Secuencia? ¿Por qué?

Ejercicio 2

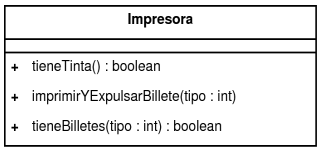
Utilizando el paradigma de orientación a objetos se desea diseñar el *software* de control de una máquina expendedora de billetes de metro. La máquina será capaz de expender diferentes tipos de billetes (tipo 1, tipo 2, tipo 3...) que se imprimen en

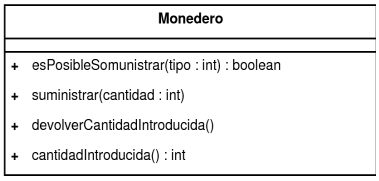
diferentes tipos de papel (tipo a, tipo b, tipo c...). Además cada tipo de billete tendrá un nombre y un precio diferente. Debiendo ser todas estas características de los billetes que expende la máquina configurables desde un menú apropiado.

Para ser capaz de realizar sus funciones la máquina dispone de dos elementos *hardware* específicos: una impresora de billetes y un monedero automático.

La clase `Impresora`.- Para controlar la impresora de billetes el fabricante de la misma suministra una biblioteca que consta de la clase `Impresora`. Esta clase permite controlar si la impresora tiene tinta y si tiene billetes de un tipo de papel determinado. Además permite ordenar a la impresora la impresión, grabación magnética y expulsión de un billete de un determinado tipo. El tipo de billete que se imprima corresponderá a un valor entero que será grabado en la banda magnética del billete para que sea reconocido por los torniquetes de la red de metro.

La clase `Monedero`.- El fabricante del monedero automático también suministra una biblioteca. Ésta consta de la clase `Monedero` que permite controlar el monedero automático. La clase permite verificar el dinero que un cliente ha introducido por la ranura y si el monedero puede devolver cierta cantidad de dinero. Además permite ordenar al monedero suministrar cierta cantidad de dinero y devolver el dinero que ha entrado por la ranura.





Esquema de funcionamiento.- Normalmente la máquina presentará un dialogo a los clientes para que éstos elijan el tipo de billete que desean comprar. Después la máquina pedirá que se introduzca el dinero y que se pulse *ENTER* cuando se termine de introducir el dinero. Seguidamente la máquina efectuará las operaciones necesarias para imprimir el billete y devolver el cambio o devolver el dinero, volviendo finalmente al punto inicial. La máquina también podrá presentar un mensaje de servicio no disponible debido a falta de tinta o billetes. Por otro lado, la máquina también deberá permitir entrar en modo mantenimiento para que un operador pueda introducir nuevos tipos de billetes, modificar o eliminar los tipos actuales, o manipular la impresora (para recargarla) y el monedero (para recoger o introducir dinero).

Se pide:

a) Proponer las clases necesarias que, añadidas a la clase `Monedero` y a la clase `Impresora`, permiten construir el expendedor de billetes. Justificar a grandes rasgos para qué sirve cada una de las clases propuestas.

b) Dibujar el Diagrama Estático de Clases presentando las clases propuestas, incluyendo los métodos y propiedades que a priori se consideren interesantes (indicando su visibilidad). Presentar en este diagrama también la clase monedero y la clase impresora y las relaciones de asociación, dependencia y herencia que a priori existan entre todas (precisando su cardinalidad cuando corresponda).

c) Dibujar un Diagrama de Secuencia en el cual se refleje el escenario en el que un cliente interactúa con la máquina para obtener un billete, introduciendo dinero de sobra, y la operación se realiza con éxito, devolviendo el cambio la máquina.

Ejercicio 3

Diseñar un programa que permita a dos personas jugar una partida de tres en raya. El juego debe desarrollarse de manera que los jugadores, en turnos alternados, dispongan tres fichas cada uno sobre un tablero de 3x3 casillas. Luego, el juego también sigue turnos alternados, de forma que los jugadores pueden mover una ficha cada vez entre casillas dos adyacentes. El juego finaliza cuando un jugador consigue disponer sus tres fichas en línea.

Ejercicio 4

Implementar el código correspondiente al ejercicio anterior.

Ejercicio 5

Ampliar el diseño del ejercicio 3 para que permita a dos personas jugar una partida de ajedrez usando el ordenador como instrumento de la partida. Se debe intentar potenciar la construcción de elementos reutilizables que puedan servir para implementar más juegos similares (como las Damas, el *Reversi*, el Tres en Raya...).

Capítulo 6 Entrada/salida en Java

El paquete `java.io` contiene un conjunto de clases que posibilitan las operaciones de entrada salida. En particular, en este capítulo, veremos el soporte que proporciona para el acceso a dispositivos permanentes de almacenamiento y para la serialización.

Para la gestión de los flujos de datos (ficheros, periféricos...) *Java* define un mecanismo muy potente conocido como ***stream***. Un *stream* es un flujo ordenado de datos que tienen una fuente y un destino. La potencia que aportan los *streams* se fundamenta en que por su estructura permiten ser concatenados incrementando su funcionalidad. Por ejemplo, es posible abrir un *stream* desde un fichero, concatenarlo a un *stream* que proporciona un *buffer* para hacer más eficiente la lectura, y concatenarlo a otro *stream* que va seleccionando las palabras que están separadas por espacios. De esta forma se consigue por ejemplo que un programa que ha abierto un fichero para lectura reciba una secuencia de palabras en vez de una secuencia de *bytes*.

Los *streams*, que *Java* proporciona para gestionar la entrada y salida de datos desde los dispositivos habituales, se encuentran definidos dentro de los paquetes contenidos en `java.io`.

Atendiendo a la naturaleza de los datos que manejan los *streams* de `io` se dividen en dos grupos:

- *Streams* de *bytes*.
- *Streams* de caracteres.

Por otro lado, atendiendo a la dirección de los datos los *streams* de `io` se dividen en dos grupos:

- *Streams* de entrada.
- *Streams* de salida.

Así pues, podemos tener *streams* de *bytes* de entrada, *streams* de *bytes* de salida, *streams* de caracteres de entrada y *streams* de caracteres de salida.

6.1. *Streams* de *bytes*

Cada dato que fluye a través de un *stream* de *bytes* es gestionado por la interfaz como un entero de 32 bits, usando solo el rango comprendido entre 0 y 255 para el valor del dato, y utilizando el valor -1 como señal de fin del flujo.

Los *streams* de *bytes* pueden ser de entrada o de salida. Los *streams* de entrada derivan de la clase abstracta `InputStream`, y los de salida de `OutputStream`. Los diagramas estáticos de la Figura 50 ilustran algunas de las clases que derivan de `InputStream` y `OutputStream`.

Los métodos principales de `InputStream` son: `read()`, `close()`, `available()`, `skip()`, `mark()`, `reset()`.

- `read()` permite leer cada vez un *byte* o un conjunto de *bytes* mediante un *array*.
- `close()` cierra el flujo impidiendo cualquier lectura posterior.
- `avaiable()` devuelve el número de elementos que se pueden leer del *stream* sin dejarlo bloqueado.
- `skip()` permite saltar la lectura de un número variable de *bytes*.
- `mark()` y `reset()` permiten marcar la posición actual de lectura sobre el *stream* y volver a ella más tarde.

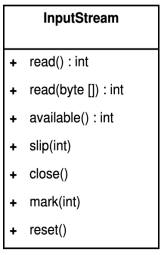


Figura 49.- Representación de la clase `InputStream` con algunos de sus miembros.

Todas las clases que derivan de `InputStream` permiten a un programa en *Java* obtener datos desde algún dispositivo o bien transformar datos que provengan de otro `InputStream`. Derivando de `InputStream` el paquete `java.io` define:

- `FileInputStream` que permite leer desde fichero.
- `ByteArrayInputStream` y `StringBufferInputStream` que permite obtener un flujo de un *array* y de un `String` respectivamente.
- `PipedInputStream` que permite recibir datos que se estén escribiendo en un tubo (*pipe*) normalmente desde otro hilo de ejecución.
- `ObjectInputStream` permite leer un objeto que ha sido serializado.
- `FilterInputStream` permite transformar los flujos procedentes de otras fuentes. Esta clase es abstracta, y de ella derivan una serie de clases que constituyen filtros concretos. Así:
  - `BufferedInputStream` interpone un *buffer* entre un *stream* de entrada y un usuario del flujo para evitar leer los *bytes* de uno en uno, acelerando la lectura.
  - `DataInputStream` transforma un flujo de *bytes* en un flujo de tipos primitivos.
  - `ZIPInputStream` y `GZIPInputStream` permiten obtener datos en claro de flujos de datos comprimidos. Se encuentran en el paquete `java.util`.

La clase abstracta `OutputStream` tiene los siguientes métodos: `close()`, `flush()` y `write()`. Estos métodos permiten respectivamente: cerrar un *stream*, volcar los datos que quedan pendientes y escribir un *byte* o *array* de *bytes*. Además, de `OutputStream` derivan clases análogas a las de entrada pero con el flujo en sentido opuesto. Es decir, clases que tienen por objeto exportar en forma de *bytes* datos de un programa hacia otro lugar.

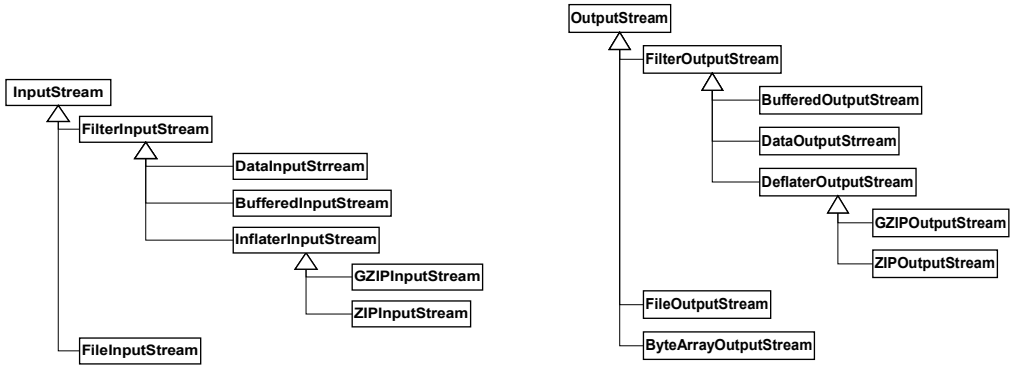


Figura 50.- Diagrama con los principales *streams* de entrada y salida de *bytes*.

El siguiente fragmento de código ilustra la apertura de un *stream* desde fichero y la lectura de los datos del mismo. Para ello se utiliza un objeto de la clase `FileInputStream` y la clase `VectorDinamico` definida en 3.2.1 para almacenarlos.

```
InputStream in = new FileInputStream("/home/user/NombreFichero");
VectorDinamico vector = new VectorDinamico();
int cont = 0;

Integer dato = in.read();
if (dato != -1)
    vector.poner(cont++,dato);

while(dato != -1) {
    dato = in.read();
    vector.poner(cont++,dato);
}
```

Los siguientes diagramas ilustran algunos de los métodos de los *streams* de *bytes* que tratan la compresión.

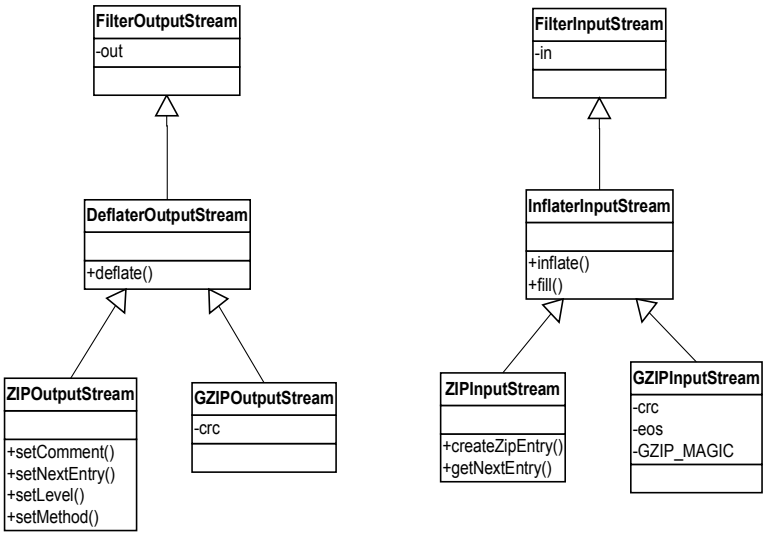


Figura 51.- *Streams* de entrada y salida de *bytes* comprimidos con algunos de sus miembros.

6.2. *Streams* de caracteres

Los datos que viajan a través de un *stream* de caracteres corresponden a sucesiones legibles de caracteres *Unicode* (de 16 bits binarios). Estos datos se gestionan en la interfaz de *stream* usando enteros de 32 bits. Los datos sólo toman valores entre 0 y 65535, y el valor -1 se usa para indicar el fin de flujo.

Los *streams* de caracteres pueden ser de entrada o de salida. Los *streams* de entrada derivan de la clase `Reader`, y los de salida de `Writer`. Los Diagramas Estáticos de Clases de la Figura 52 ilustran algunas de las clases que derivan de `Reader` y `Writer`.

Todas las clases que derivan de `Reader` tienen por objeto introducir caracteres en un programa *Java* desde otro sitio. Las principales clases derivadas de `Reader` son:

- `BufferedReader` para construir un *buffer* que mejore el rendimiento en el acceso a otro flujo de entrada. Además añade el método `readLine()` que devuelve un `String` por cada invocación.
- `CharArrayReader` y `StringReader` para transformar *arrays* de caracteres y objetos `String` en flujos de caracteres respectivamente.
- La clase `InputStreamReader` es un puente entre un `InputStream` y un `Reader`, que permite transforma flujos de *bytes* en flujos de caracteres.
- La clase `FileReader` derivada de `InputStreamReader` permite leer caracteres de fichero.

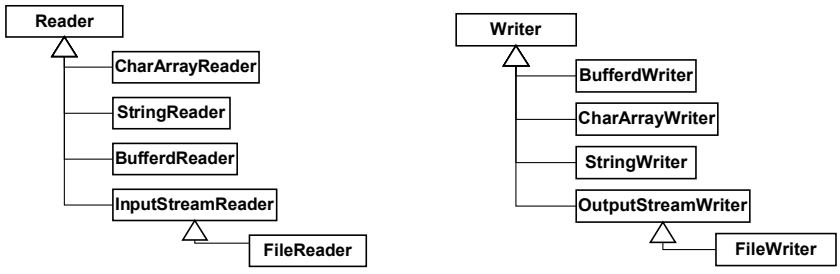


Figura 52.- Diagrama con los principales *streams* de entrada y salida de caracteres.

De la clase abstracta `Writer` derivan clases análogas a las de entrada pero en sentido opuesto. Su objetivo es exportar en forma de caracteres los datos producidos por un programa *Java*.

El siguiente ejemplo crea un `FileReader` que lee un fichero carácter a carácter.

```
char cr;
Reader in = new FileReader("Nombre fichero");
do {
   cr = in.read();
} while (cr != -1);
```

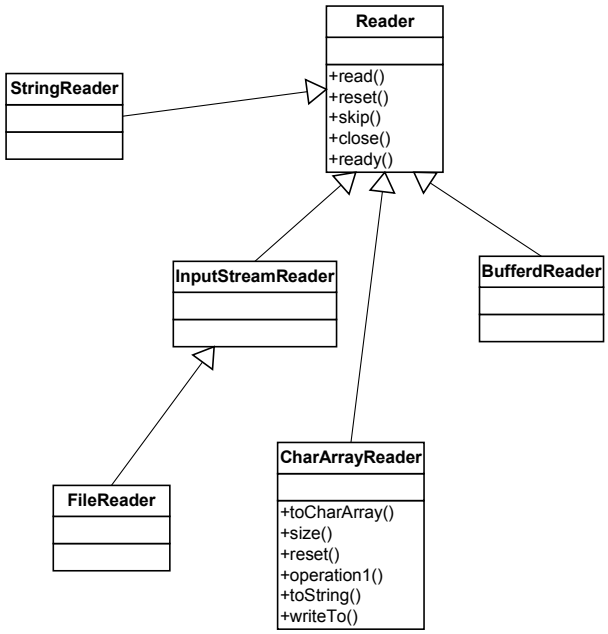


Figura 53.- *Steams* de entrada de caracteres con algunos de sus miembros.

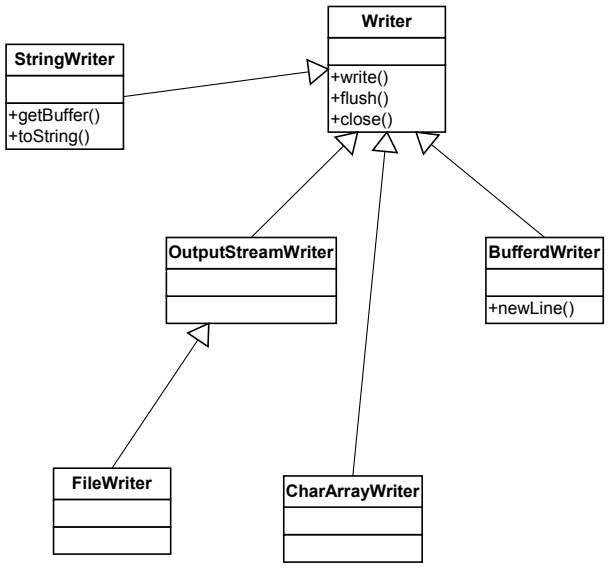


Figura 54.- *Streams* de salida de caracteres con algunos de sus miembros.

Las operaciones de lectura y escritura en un fichero utilizando directamente `FileReader` y `FileWriter` pueden ser muy ineficaces porque en cada operación de entrada/salida sólo se lee un *byte* y se puede desaprovechar la capacidad del periférico de tratar una serie de datos de una vez. Para agilizar el proceso se debe utilizar un *buffer* que permita tratar un conjunto de *bytes* en cada operación de entrada/salida. El tamaño del *buffer* debe ajustarse teniendo en cuenta la capacidad de almacenamiento en memoria principal del ordenador y las necesidades de lectura o escritura. Las clases `BuffreReader` y `BufferWriter` pueden ser concatenadas a `FileReader` y `FileWriter` respectivamente, y permiten definir el tamaño del *buffer* que se utilizará en cada operación de entrada/salida. A continuación se presenta un ejemplo en el que se utiliza un *buffer* de 10 datos para leer de disco. Con ello las operaciones físicas contra disco se reducen en media a 1 de cada 10.

```
int c;
Reader in = new FileReader("C:\\prueba.txt");
Reader buf = new BufferedReader(in,10);
do {
    c = buf.read();
    char cr = (char) c;
    System.out.print(cr);
} while (c != -1);
```

6.2.1 *Streams* del sistema

Unix introdujo el concepto de entrada/salida estándar para definir los flujos de entrada, salida y error que por defecto utiliza un programa. *Unix* definió tres flujos: la entrada, la salida y la salida de error. Por eso en *Java* existen esos mismos tres *streams* de *bytes*:

- `System.in`
- `System.out`
- `System.err`

A lo largo de los temas precedentes hemos utilizado `System.out` para mostrar mensajes por pantalla. Ahora sabemos que `System.out` no es la pantalla sino la salida estándar, que por defecto es la pantalla. En caso de que la salida de nuestro programa en *Java* estuviese conectada (mediante un *pipe* del *shell*) con otro programa los datos enviados a `System.out` se dirigirían a la entrada de ese programa.

De la misma forma, los datos que se envíen al flujo `System.err` se dirigirán a la salida de error estándar que tenga definido nuestro sistema operativo.

El flujo de entrada `System.in` es más difícil de usar. `System.in` es un `InputStream`, por lo que los elementos que devuelve son *bytes*. Cuando se hace `read` sobre él, la ejecución se suspende hasta que por la entrada estándar entra un flujo de *bytes*. En el caso del teclado esto ocurre al pulsar una serie de teclas y finalmente la tecla `Enter`.

```
int c = System.in.read();
System.out.println(c);
```

Si queremos utilizar `System.in` para leer caracteres lo más sencillo es transformarlo en un `Reader` (ya hemos visto que eso es posible utilizando la clase `InputStreamReader`). Además es conveniente conectar este flujo a `BufferedReader` para que nos devuelva cada cadena introducida por teclado como un `String`, en vez de como una sucesión de números. El siguiente ejemplo define un `Reader` sobre `System.in` y le acopla un `BufferedReader` para leer una cadena del teclado y luego imprimirla por pantalla.

```
Reader i = new InputStreamReader(System.in);
BufferedReader r = new BufferedReader(i);
String s = r.readLine();

System.out.print("Ha escrito: ");
System.out.println(s);
```

6.2.2 `StreamTokenizer` y `Scanner`

La clase `StreamTokenizer` permite transformar un `InputStream` en una secuencia de *tokens* más fácilmente utilizable por un programa. Los *tokens* por defecto son palabras o números separados por espacios.

Cuando se le envía un mensaje de `nextToken()` a un `StreamTokenizer` se queda esperando a que lleguen *bytes* por el flujo que se le ha asociado en su creación. Conforme va llegando ese flujo se espera a que se pueda completar una palabra o un número. En cuanto se tiene una de las dos, la ejecución continúa con la siguiente instrucción a `nextToken()`.

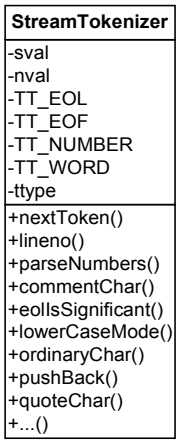


Figura 55.- Algunas de las propiedades de la clase que divide un *stream* en *tokens*.

El siguiente ejemplo ilustra el uso de un `StreamTokenizer` para leer una cadena desde teclado, leyendo y distinguiendo cada vez números o palabras.

`StreamTokenizer` permite seleccionar el carácter que se utiliza como separador de *tokens*, como fin de línea e incluso elegir un carácter para identificar comentarios que no se asocian a ningún *token*. Cuando el *stream* proviene de fichero la condición de *EOF* (*end of file*) se produce de manera natural, sin embargo cuando el *stream* proviene del teclado es necesario forzar EOF mediante la combinación CRTL+C.

```
int resultado=0;
StreamTokenizer nums = new StreamTokenizer(System.in);

while (nums.nextToken() != StreamTokenizer.TT_EOF) {
    if (nums.ttype ==  StreamTokenizer.TT_WORD)
        System.out.println(nums.sval);
    else if (nums.ttype ==  StreamTokenizer.TT_NUMBER)
        System.out.println(nums.nval);
}
```

La clase `Scanner` se califica habitualmente como el relevo de `StreamTokenizer`. `Scanner` permite definir la expresión regular que tokeniza el *stream*. La profundidad de las expresiones regulares está más allá del objeto de este texto, por lo que se recomienda la lectura de un texto específico.

6.3. Manejo de ficheros

Java, además de los *streams* proporciona tres clases más para el manejo de ficheros:

- `File`.- Esta clase proporciona métodos útiles para trabajar con ficheros (crear directorios, renombrarlos, consultar el tamaño, borrar ficheros y directorios, listar elementos de un directorio, etc.).
- `PrintWriter`.- Clase que permite escribir en un fichero utilizando una sintaxis similar a la de la función `printf()` de C.
- `RandomAccessFile`.- Esta clase permite tratar un fichero como un gran *array* de caracteres de entrada y salida al que se tiene una referencia.

Los siguientes diagramas describen los principales miembros de las clases `File` y `RandomAccessFile`.

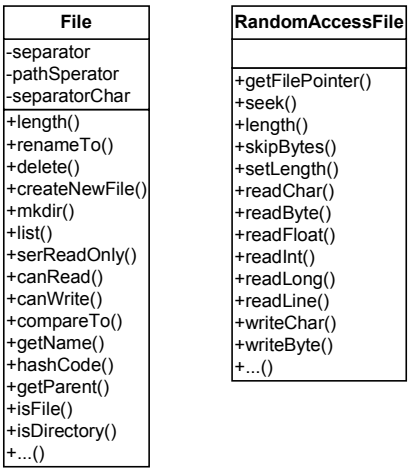


Figura 56.- Algunas de las propiedades de clases útiles para manejo de ficheros.

6.4. La interfaz Serializable

La persistencia se ha descrito en el primer capítulo como una de las capacidades que debería proporcionar un lenguaje de programación orientado a objetos. *Java* da soporte a la persistencia mediante la **serialización**. Se llama serialización al proceso de convertir un objeto en un flujo de *bytes* y se llama **deserialización** a la reconstrucción del objeto a partir del flujo de *bytes*.

Para permitir un proceso de serialización común a todas las clases de objetos *Java* proporciona la interfaz `Serializable`. Esta interfaz no define métodos y sólo sirve para indicar a los métodos de la clase `ObjectOutputStream` que pueden actuar sobre aquel que la implemente. La clase `ObjectOutputStream` define el método `writeObject()`, que permite convertir un objeto serializable en un flujo de datos de salida. Por otro lado, la clase `ObjectInputStream` permite transformar un flujo de datos de entrada en un objeto gracias al método `readObject()`.

Los objetos se serializan llamando a `writeObject()` y se deserializan con el método `readObject()`. Casi todas las clases definidas en los paquetes estándar de *Java* se pueden serializar (por ejemplo `String`, los envoltorios...). Básicamente, la

serialización de una clase consiste en enviar a un flujo sus propiedades no estáticas, y la deserialización el proceso contrario. Para evitar que una propiedad determinada se serialice (por ejemplo una contraseña) se puede utilizar en su declaración la etiqueta de prohibición de serialización `transient`.

Para que una clase pueda serializarse, todas las propiedades contenidas en la clase deben corresponder a tipos primitivos o a objetos cuyas clases cumplan a su vez la interfaz serializable. En otro caso, al ejecutar `writeObject()` obtendremos una excepción del tipo `NotSerializableException` de `java.io`. Por eso, en el ejemplo siguiente, tanto la clase `Coche`, como la clase `Rueda`, utilizada por `Coche`, son serializables.

Cuando un objeto se deserializa no se obtiene el mismo objeto sino una copia de aquél que se serializó. Es decir, tanto sus propiedades como su comportamiento son iguales, pero no conserva la misma identidad. El siguiente ejemplo ilustra este aspecto.

```
class Rueda implements Serializable {
    public int presion = 1;
}
class Coche implements Serializable {
  public int velocidad;
  public Rueda r1 = new Rueda();
  public Rueda r2 = new Rueda();
  public Rueda r3 = new Rueda();
  public Rueda r4 = new Rueda();
}
public class Ejemplo {
   public static void main(String[] args) {
     try {
       Coche c = new Coche();
       c.r1.presion = 24;
       String fic = "c:\\Coche.txt";
       ObjectOutputStream salida = new ObjectOutputStream(new FileOutputStream(fic));
       c.velocidad = 2;
       c.r1.presion = 3;
       salida.writeObject(c);                    //Se escribe el objeto a disco
       System.out.println(c.r.presion);          //La presion valdrá 3
       salida.close();

       ObjectInputStream entrada = new ObjectInputStream(new FileInputStream(fic));
       Coche c_copia = (Coche) entrada.readObject();
       System.out.println(c_copia.velocidad);      //Estas líneas imprimirán un 1 y
       System.out.println(c_copia.r1.presion);     //La presión valdrá 24 porque el valor
     }                                             //se guardó antes de cambiarlo a 3
     catch(Exception e) {
       System.out.println(e);
     }
  }
}
```

Si se desea tener un control mayor sobre el proceso de serialización se puede implementar la interfaz `Externalizable`, en vez de `Serializable`. Esta interfaz, derivada de `Serializable`, permite definir los métodos `writeExternal()` y `readExternal()`. Estos métodos se invocan automáticamente durante la serialización y la deserialización para permitir al programador controlar el proceso.

6.5. Conclusiones

En este capítulo se ha visto que *Java* proporciona dos mecanismos básicos de manejo de ficheros, uno de acceso aleatorio y otro de acceso mediante flujos. Será tarea del programado decidir qué mecanismo es más conveniente en cada caso.

Además, proporciona mecanismos de serialización y deserialización que simplifican la tarea de dotar de persistencia a las aplicaciones que se desarrollen.

6.6. Ejercicios

Ejercicio 1

Con objeto de sustituir en un aeropuerto unas pantallas analógicas de Radar, se desea construir un programa en *Java* que permita gestionar una pantalla digital de Radar.

Un fabricante de pantallas digitales ha proporcionado unas pantallas táctiles de gran tamaño que vienen acompañadas de un *software* de control de las mismas. Este *software* consta de la clase `RadarScreen` y de la interfaz `RadarButton`.

Los métodos de la clase `RadarScreen`, orientados a la interacción con las pantallas, son:

`void radarScreen(double latitud, double longitud)`.- Constructor donde se le indica a la pantalla la latitud y la longitud de la ubicación del radar. De esta forma la pantalla conoce automáticamente el mapa que debe presentar.

`void draw(int color, double latitud, double longitud, String etiqueta)`.- Este método permite dibujar en la pantalla un objeto. La propiedad color indica al sistema el color que tendrá el objeto que se represente y que será distinto para cada entero. Las propiedades latitud y longitud permiten especificar la latitud y la longitud del objeto que se desea representar. Por último, la propiedad etiqueta permite asociar un texto a este objeto para que se visualice junto a él.

`void clearScreen()`.- Limpia la pantalla, preparándola para pintar los objetos.

`void setButton(RadarButton button)`.- Permite añadir un botón a la pantalla. El botón será cualquier objeto que implemente la interfaz `RadarButton`.

La interfaz `RadarButton` define dos métodos:

`String label()`.- Devuelve la etiqueta que mostrará la pantalla al pintar el botón.

`click()`.- Esté método será llamado por la pantalla cuando alguien pulse el botón asociado. La implementación de este método contendrá el código que implementará el comportamiento que se desee para el botón.

A su vez, el fabricante del radar ha proporcionado una clase llamada `RadarStreamReader` que permite la lectura de los datos transmitidos por el radar. Esta clase se puede concatenar a un `StreamTokenizer` para obtener los datos sobre los aviones que el radar detecta. Es decir, se puede ir pidiendo el valor (`nval` o `sval`) a cada `Token` (con `nextToken()`) de manera indefinida.

El radar tiene un comportamiento cíclico, que se repite a intervalos regulares. Por eso, los datos proporcionados por la clase `RadarStreamReader` también son cíclicos. Debiéndose pintar los datos en cada ciclo, y borrar la pantalla al principio del ciclo siguiente.

Los datos que proporciona la clase se repiten según la siguiente secuencia: identificador, tipo, latitud, longitud, etiqueta.

`identificador`.- Identificador del objeto detectado por el radar. Será un entero mayor o igual a cero. El primer objeto volante será el 0. El siguiente el 1. Y así sucesivamente hasta que vuelva a aparecer el elemento con identificador 0.

`tipo`.- Entero que identifica los diferentes tipos de objetos volantes (`militares = 1`, `civiles = 2` y `no_identificados = 3`). Cada tipo deberá presentarse de un color diferente en la pantalla, que se corresponderá con el número asociado al tipo.

`latitud` y `longitud`.- Latitud y longitud del objeto detectado.

`etiqueta`.- Cadena identificativa (`String`) transmitida por el objeto volante al radar para su presentación en pantalla.

Se pide:

a) Presentar las clases que a priori se crean necesarias para resolver el problema del bucle cíclico de presentación de objetos volantes (incluidas las clases que vienen impuestas por los fabricantes y las nuevas que se consideren). Explicar la razón de ser (el contrato) de las clases nuevas que se incluyan. Dibujar los Diagramas Estáticos de Clases que presenten las relaciones entre todas las clases, incluyendo propiedades, métodos y visibilidad de cada uno de ellos.

b) Dibujar un Diagrama de Secuencia con el escenario del bucle de presentación de imágenes (sólo es necesario presentar una iteración del bucle). Escribir en *Java* el código de todos los métodos involucrados en este diagrama.

c) Se desea añadir un botón para detectar las colisiones entre los objetos volantes presentes en la pantalla. Para ello, se ha comprado una nueva clase para detectar colisiones. Esta clase es un s*tream* llamado `CollisionDetectorStreamReader` y tiene un constructor al que se le pasa como parámetro un `RadarStreamReader`. Por lo demás se comporta igual a `RadarStreamReader` excepto por que el tipo asignado a los objetos que pueden colisionar se pone a 0. Por eso, se desea que al apretar este botón de detección de colisiones, los objetos que pueden colisionar cambien a color 0. Detallar las clases nuevas que serán necesarias para lograr este comportamiento. Mostrar en un Diagrama Estático de Clases las relaciones con las clases e interfaces existentes. Mostrar en un Diagrama de Secuencia el comportamiento dinámico desde que se aprieta el botón hasta que se inicia el bucle cíclico de presentación de objetos volantes.

Capítulo 7 Estructuras de datos predefinidas en Java

Aunque en *Java* la clase que se utiliza como contenedor básico es el *array* (ya descrito en el capítulo 2), el paquete `java.util` añade toda una jerarquía de interfaces y clases enfocadas a permitir la creación de objetos contenedores de otros objetos. Estos contenedores permiten almacenar objetos utilizando diferentes políticas de almacenamiento.

Los contenedores que se encuentran en el paquete `java.util` pueden ser de dos tipos básicos: tipo `collection` y tipo `map`.

Hasta la versión 1.5 los contenedores no utilizaban genericidad y por lo tanto no conocían el tipo del objeto que contenían. A partir de la versión 1.5 todos los contenedores permiten la parametrización del tipo que contienen, aunque por razones de compatibilidad se ha mantenido también la versión sin genericidad.

7.1. `Collection`

Todas las clases que implementan la interfaz `Collection` se caracterizan por constituir grupos de objetos individuales con una relación de orden entre ellos. Todos los objetos de tipo `Collection` comparten los siguientes métodos: `add(objeto)`, `isEmpty()`, `contains(objeto)`, `clear()`, `remove()`, `size()`, `toArray()` e `iterator()`.

Entre los derivados de `Collection` se pueden destacar:

`Set`.- Interfaz que puede contener un conjunto de elementos donde no se admiten repeticiones.

`SortedSet`.- Interfaz que deriva de `Set` cuyos elementos están ordenados. Añade los siguientes métodos: `first()`, `last()`, `subSet()`, `headSet()`, `tailSet()`. Precisa que los elementos insertados cumplan la interfaz `Comparable` y utiliza la comparación que esta interfaz define para determinar si dos objetos son iguales. Los objetos, una vez insertados no pueden cambiar su relación de orden o los resultados serán imprevisibles.

`TreeSet`.- Clase que implementa `SortedSet` en la que los elementos internamente se mantienen ordenados mediante una estructura de árbol.

`HashSet`.- Clase que implementa `Set` con una tabla *hash*. Como ventaja aporta que el número de objetos contenidos en la estructura no repercute en la velocidad de las operaciones de búsqueda, eliminación e inserción. Además los objetos pueden cambiar libremente pues la relación de orden no depende de su estado. La desventaja estriba en que no se pueden recorrer según un orden fijado por el programador. Utiliza el método `hashCode()`, definido en la clase `Object`, para determinar si dos objetos son iguales.

`List`.- Interfaz de elementos ordenados. Aporta los siguientes métodos a la interfaz `Collection` de la que deriva: `get(índice)`, `set(índice,objeto)`, `add(índice,objeto)`, `remove()`, `indexOf()`, `subList(min,max)`.

`ArrayList`.- Clase que implementa `List` mediante un *array*. Tiene la ventaja de permitir un acceso aleatorio rápido a los elementos que contiene. La inserción tiene un coste muy bajo, excepto cuando se sobrepasa el tamaño actualmente reservado, momento en el que automáticamente se debe aumentar el tamaño y copiar su contenido. Las operaciones de inserción y eliminación, si no son por el final, tienen un coste elevado, ya que implican movimiento de memoria.

`LinkedList`.- Clase que implementa `List` usando un conjunto de nodos doblemente enlazados. El acceso aleatorio a los elementos de esta estructura es más lento que en el caso del `ArrayList`. Sin embargo, la inserción de elementos se realiza en tiempo constante. Aporta los siguientes métodos: `getfirst()`, `getLast()`, `addFirst(objeto)`, `addLast(objeto)`, `removeFirst()`, `removeLast()`.

`Vector`.- Clase que implementa `List` y que permite construir vectores. Entre otros métodos añade algunos de acceso aleatorio como: `elementAt()` e `insertElementAt()`. `Vector` es similar a `ArrayList` salvo por dos diferencias importantes. `Vector` está diseñada para un uso concurrente seguro y `ArrayList` no, por lo que `Vector` es más segura pero ofrece peor rendimiento. Además, `Vector` permite especificar cómo crece su capacidad cuando se agota su capacidad actual.

`Stack`.- Clase que extiende `Vector` para permitir usarla con las operaciones habituales en pilas. Añade métodos como `push` y `pop` para insertar y obtener elementos de la pila.

Ejemplo de uso de `ArrayList`

El siguiente ejemplo ilustra el uso de la clase `ArrayList` para almacenar objetos de la clase `String` sin usar tipos genéricos.

```
List lista = new ArrayList();
for (int cont = 0; cont < 100; cont++)
    lista.add("Hello");

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

A continuación se presenta el mismo ejemplo usando tipos genéricos.

```
List <String> lista = new ArrayList<String>();
for (int cont = 0; cont < 100; cont++)
    lista.add("Hello");

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

Obsérvese que el tipo de la variable suele definirse usando la interfaz, independizando de esta manera el código de la implementación particular que se elija.

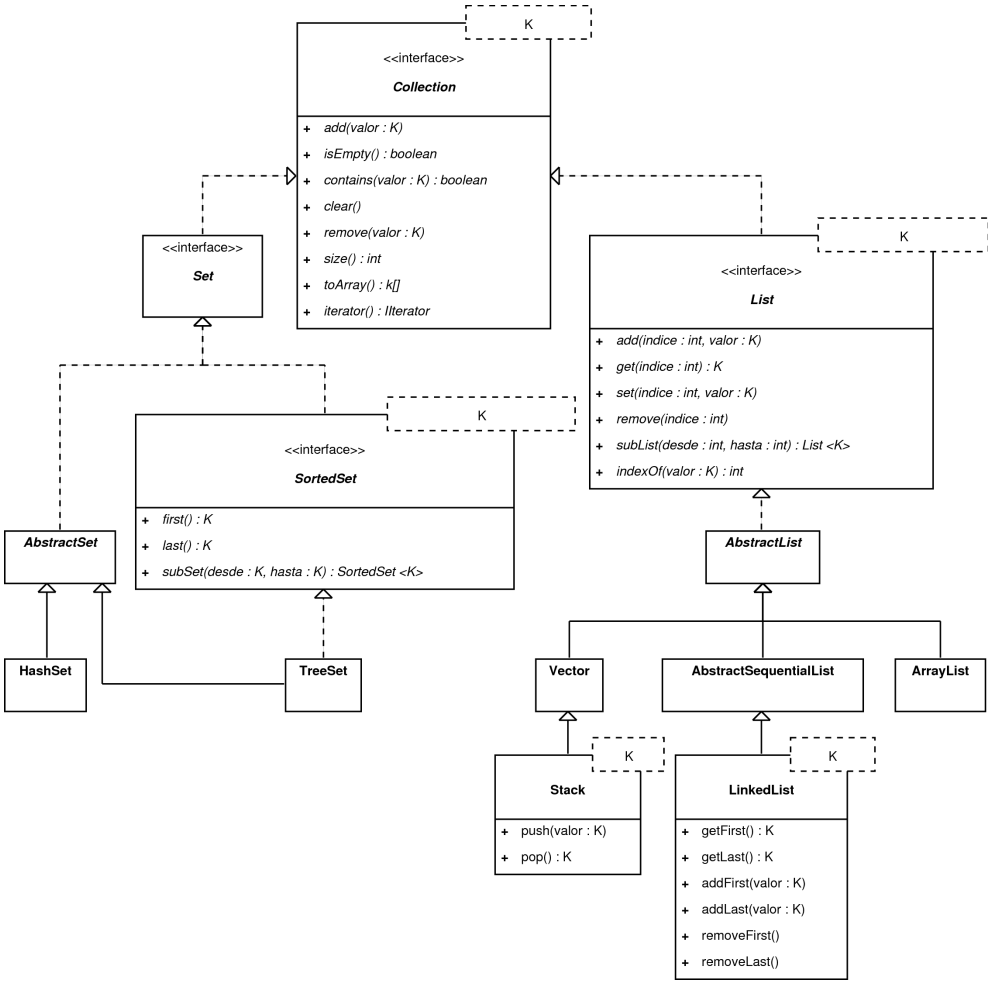


Figura 57.- Diagrama con los principales contenedores derivados de `Collection`.

7.1.1 El `EnumSet`

Ya se comentó, cuando se habló de tipos enumerados, que un enumerado es un conjunto de objetos finito y conocido y, como tal, puede agruparse en una `Collection`. Sin embargo, dado que el conjunto de objetos generado en un enumerado está muy controlado, se ha creado un contenedor optimizado para ellos. Dicho contenedor es el `EnumSet`.

El `EnumSet` es una implementación especial de la interfaz `Set`. Todos los elementos contenidos en un `EnumSet` deben pertenecer al mismo tipo de enumerado, indicando dicho tipo en la construcción del `EnumSet`. Esta implementación es extremadamente compacta y eficiente, siendo el reemplazo del tradicional *array* de bits que se solía utilizar antes. Veamos un ejemplo de uso del `EnumSet`.

```
//Creamos un conjunto vacio de DiasSemana
EnumSet<DiasSemana> diasVacio = EnumSet.noneOf(DiasSemana.class);

//Añadimos el miercoles
diasVacio.add(DiasSemana.MIERCOLES);

//Creamos un conjunto con todos los dias de la semana
EnumSet<DiasSemana> diasSemana = EnumSet.allOf(DiasSemana.class);

//Recorremos el EnumSet con un bucle for-each
for (DiasSemana dia : diasSemana) {
    System.out.println(dia.toString());
}
```

7.2. Map

Los contenedores de la clase `Map` se caracterizan por constituir grupos de pares de objetos clave-valor, de forma que todo valor tiene asociado al menos una clave. Por ejemplo, un vector puede verse como un `Map` en los que la clave es un entero (el índice) y los valores cualquier tipo de objetos. Los `Map` generalizan este concepto permitiendo que también el tipo del índice pueda ser cualquier clase de objetos. Por ejemplo, para construir un directorio telefónico, se podría utilizar un `Map` que use para la clave el nombre y para el valor un entero con el número telefónico.

`Map`.- Los objetos que cumplen la interfaz `Map` tiene los siguiente métodos: `size()`, `isEmpty()`, `containsKey(objecto)`, `containsValue(objeto)`, `get(objeto)`, `put(objeto,objeto)`, `remove(objeto)`, `clear()`.

`SortedMap`.- Interfaz derivada de `Map` que obliga a que los elementos que lo constituyen cumplan la interfaz `Comparable`.

`TreeMap`.- Clase que implementa `SortedMap` con un `TreeSet` para los índices.

`HashMap`.- Clase que implementa `SortedMap` con `HashSet` para los índices.

Como ya hemos dicho se debe cuidar que los elementos almacenados en los `TreeSet` no cambien su criterio de comparación, ya que en ese caso el contenedor no garantiza su comportamiento. Evidentemente esto se extiende a los índices de un `TreeMap`. Cuando forzosamente se deban utilizar como índices objetos mutables que cambien su relación de orden, se deberá usar `HashMap`, en otro caso se puede preferir `TreeMap` por el orden que se obtiene al recorrerlos.

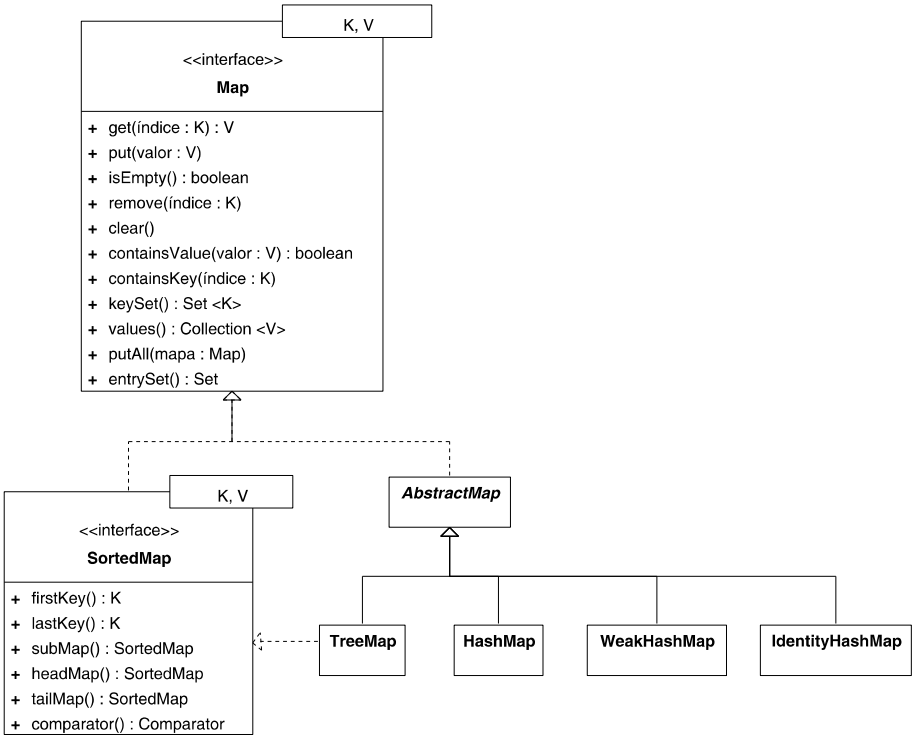


Figura 58.- Diagrama con varios contenedores derivados de `Map`.

Ejemplo de uso de `HashMap`

El siguiente ejemplo ilustra el uso de la clase `HashMap` para almacenar objetos de la clase `String`, utilizando a su vez como índice un `String`.

```
Map m = new HashMap();
m.put("Pedro Gutierrez","91 424 12 12");
m.put("Juan Gonzalez","91 434 43 12");
```

7.3. Iteradores

Los iteradores constituyen el mecanismo proporcionado por la biblioteca de *Java* para recorrer secuencialmente los elementos de cualquier contenedor. Los iteradores tienen los siguientes métodos: `hasNext()`, `next()`, y `remove()`.

El siguiente ejemplo ilustra el uso de un iterador para recorrer una lista de objetos de la clase `Coche`, aplicando luego un método sobre ellos:

```
Iterator <Coche> e = mi_lista_coches.iterator();
while (e.hasNext()) {
    Coche c = e.next();
    c.acelerar();
}
```

A partir de la versión 5 de *Java*, la estructura de control `for` permite el uso de objetos que cumplan la interfaz `iterator` para construir bucles de una manera simplificada (es lo que se denomina un bloque `for-each`).

```
    for (<parametro>:<expresion>)
       ámbito | sentencia
```

En esta estructura el identificador `expresion` corresponde a un *array* o un objeto iterable, mientras que el identificador `parametro` corresponde a una referencia de tipo compatible con los elementos que componen el *array* o el objeto iterable.

El siguiente fragmento de código suma todos los elementos de un *array* de enteros.

```
int [] array_enteros = new int [30];
int suma = 0;
for (int v:array_enteros)      {
    suma += v;
}
```

Este otro fragmento de código suma el tamaño de las cadenas contenidas en un vector.

```
Vector <String> vector_cadenas = new Vector <String>();
vector_cadenas.add("Jose");
vector_cadenas.add("Andres");
int suma = 0;

for (String str:vector_cadenas) {
    suma += str.size();
}
```

En el uso de iteradores sobre `Collections` debe tenerse precaución si la estructura de datos cambia mientras que se recorre. En particular, `ArrayList` y `Vector` por su diseño, son más susceptibles de que se produzcan errores si su estructura cambia mientras se recorren con un iterador.

7.4. La clase `Collections`

Para facilitar el uso de las colecciones se añadió a *Java* la clase `Collections`. Esta clase está formada exclusivamente por métodos estáticos que devuelven u operan sobre colecciones.

En algunas ocasiones es interesante disponer de colecciones de solo lectura. Por ejemplo, si un método devuelve un `ArrayList`, puede interesar que el receptor de dicha colección no puede modificarla. Para posibilitar este comportamiento, la clase `Collections` dispone de métodos que permiten envolver cualquier colección y devolver una versión de solo lectura, tal que si se intenta añadir un objeto a cualquiera de ellos se produce una `UnsupportedOperationException`. Estas colecciones de solo lectura también se dice que son inmutables. Una propiedad interesante de tales clases es que, al ser imposible su modificación es posible su uso simultáneo desde varios hilos sin riesgo de condiciones de carrera.

Además, la clase `Collections` posee métodos estáticos parametrizados muy útiles para simplificar la devolución de colecciones vacías. Dichos métodos son `emptyList()`, `emptySet()` y `emptyMap()`. Los objetos devueltos son inmutables.

La clase `Collections` también posee métodos para: encontrar los máximos y mínimos de los elementos contenidos en una `collection`, copiar una lista en otra, realizar búsquedas binarias sobre listas, etc.

Conocer los métodos de utilidad que ofrece esta clase puede ahorrar mucho trabajo a la hora de trabajar con colecciones y mapas y es bueno tenerla en cuenta.

7.5. El resto del paquete `java.util`

El paquete `java.util` es un pequeño cajón de sastre que contiene multitud de clases que nos hacen más fácil construir programas en *Java*. A parte de los contenedores encontramos clases para el tratamiento de fechas (`Date`, `Calendar`), para definir alarmas (`Timer`, `TimerTask`), para el tratamiento de cadenas de texto (`Scanner`, `Formatter`, `StringTokenizer`), para la generación de números aleatorios (`Random`), para utilizar elementos relativos a la configuración local del equipo (`Locale`, `TimeZone`, `Currency`), para facilitar la concurrencia (`Semaphores`, `ArrayBlockingQueue`, `SynchronousQueue`...).

7.6. Conclusiones

Se puede pensar que conocer un lenguaje de programación consiste en conocer las primitivas del mismo y poco más, pues la flexibilidad de los lenguajes de programación permite que el programador desarrolle todo cuanto necesite partiendo de cero. Además, como ya se dijo en el primer capítulo, los desarrolladores suelen preferir usar sus propios desarrollos a los construidos por terceros. Este hecho se deriva de la poca confianza que existe en los desarrollos de otras personas, y también, de los problemas de comunicación relativos al traspaso de *software*.

Para atajar la creciente complejidad del *software* este hecho debe cambiar. Conocer las bibliotecas estándar de un lenguaje resulta imprescindible para obtener resultados garantizados. Hay que entender que es mejor no crear todo partiendo de cero cada vez. La asociación y la herencia deben ser los vehículos que permitan especializar las clases proporcionadas por estas bibliotecas, adaptándolas a los comportamientos particulares que un diseño particular precisa.

7.7. Lecturas recomendadas

"El lenguaje de programación *Java*", 3ª Edición, Arnold Gosling, Addison Wesley, 2001.

"Piensa en *Java*", 2ª Edición, Bruce Eckel, Addison Wesley, 2002.

"Introducción a la Programación Orientada a Objetos con *JAVA*", C. Thomas Wu, Mc Graw Hill, 2001.

7.8. Ejercicios

Ejercicio 1

Diseñar, apoyándose en la clase `Set`, una clase `BomboLoteria` que permita construir un programa para jugar al bingo.

Capítulo 8 Patrones y principios de diseño

Ya se han revisado los principales elementos que permiten la Programación Orientada a Objetos. Sin embargo, hay que señalar que un buen diseño orientado a objetos no sólo consiste en utilizar los elementos que se han explicado sin orden ni concierto, sino que juega un papel fundamental la experiencia del diseñador. Hay que encontrar las abstracciones adecuadas, construir interfaces eficaces y establecer las relaciones necesarias entre ellas. Además, todo ello debe hacerse de manera específica para el problema que se pretende resolver, para que la programación ofrezca la ventaja de acercarse al dominio del problema. Por otro lado, y también debe hacerse de manera genérica, para que sea fácil incorporar nuevos requisitos o resolver problemas distintos con los mismos objetos.

En este tema se esboza el principio del camino que un diseñador de programas orientados a objetos debe recorrer. Para ello se introduce el concepto de Patrón de Diseño y se presentan unos cuantos de los más importantes. Posteriormente se enuncian algunos principios que ayudan a crear un buen diseño.

8.1. Principales Patrones de Diseño

Es un hecho que, en general, los diseñadores noveles no son capaces de hacer buenos diseños, pues no utilizan adecuadamente las herramientas que la Programación Orientada a Objetos proporciona. Mientras, los diseñadores experimentados se caracterizan por conocer multitud de buenas soluciones a problemas que ya han tenido que resolver, y reutilizan estas soluciones adaptándolas a los nuevos problemas que abordan.

Los Patrones de Diseño intentan capturar esa experiencia en un conjunto de diseños genéricos que sean aplicables a un sin fin de problemas. Según Christopher Alexander, destacado arquitecto del siglo XX, "cada patrón de diseño describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema de tal modo que se pueda aplicar esta solución una y otra vez, sin repetir cada vez lo mismo". Esta descripción es válida también para los Patrones de Diseño de la Programación Orientada a Objetos. En los siguientes puntos se analizan algunos de los Patrones de Diseño más utilizados.

8.1.1 Fábrica Abstracta (*Abstract Factory*)

Una de las operaciones más comunes al programar con objetos es, lógicamente, la creación de los objetos. Sin embargo, esta operación es en si misma un foco de acoplamiento. Cuando desde un fragmento de código `C` invocamos la creación de un objeto de la clase `A` estamos acoplando ese código con el de la clase `A`. Si posteriormente deseamos cambiar el objeto de clase `A` por otro de otra clase que cumpla su misma interfaz no podremos hacerlo sin cambiar el código `C`.

Para evitar este problema se suele delegar la creación de los objetos a otro objeto que se denomina fábrica (*factory*). De esta manera desde el código `C` no creamos el objeto `A` directamente, sino que se lo pedimos a la clase fábrica. Evidentemente, para evitar el acoplamiento de la fábrica con el código `C`, la propia fábrica deberá obtenerse por parámetros.

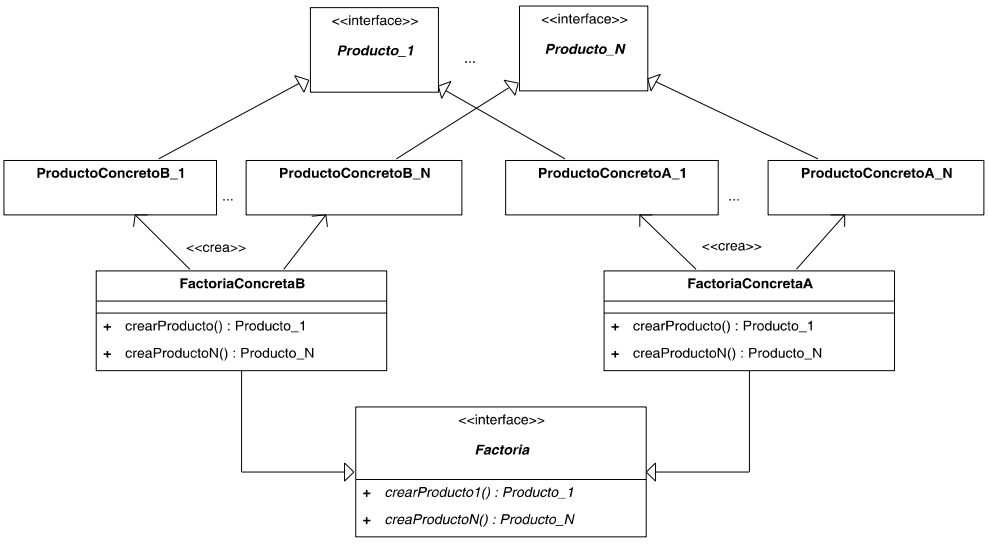


Figura 59.- Cada producto concreto se crea desde un constructor especifico.

Además, el patrón Fábrica permite ocultar los constructores de las clases que crea, de manera que solo se puedan crear los objetos a través de la fábrica, impidiendo de esta forma su creación de una forma no prevista.

A continuación se presenta un ejemplo en el que el patrón método de fabricación se utiliza para que un sistema gráfico pueda crear diferentes tipos de registros en una aplicación de gestión de piezas en un taller. En este caso, la separación entre los objetos fábrica y la aplicación de gestión permite añadir nuevos tipos de piezas y que la aplicación no deba ser modificada.

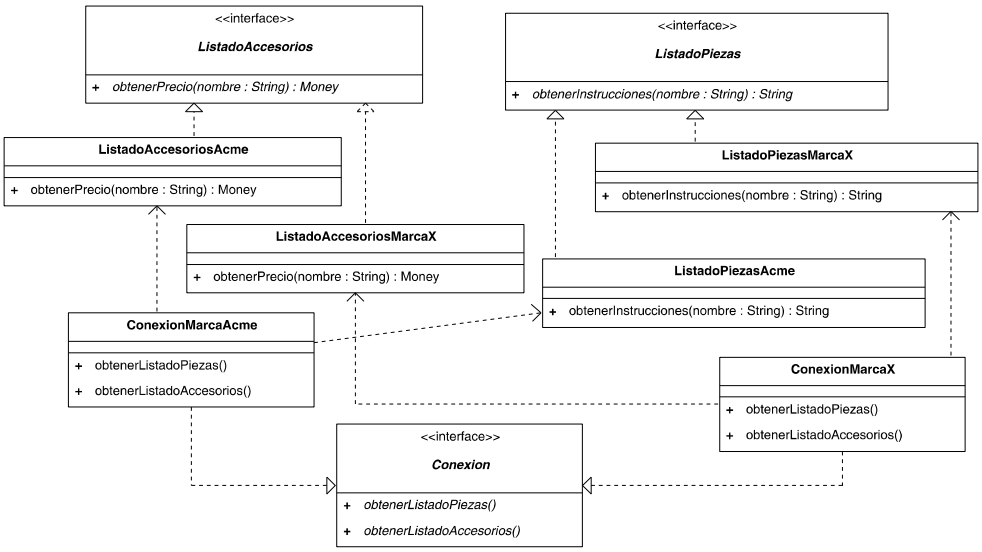


Figura 60.- En este ejemplo se dispone de un constructor que permite crear conexiones con las diferentes bases de datos de los fabricantes a un programa de facturación de un taller.

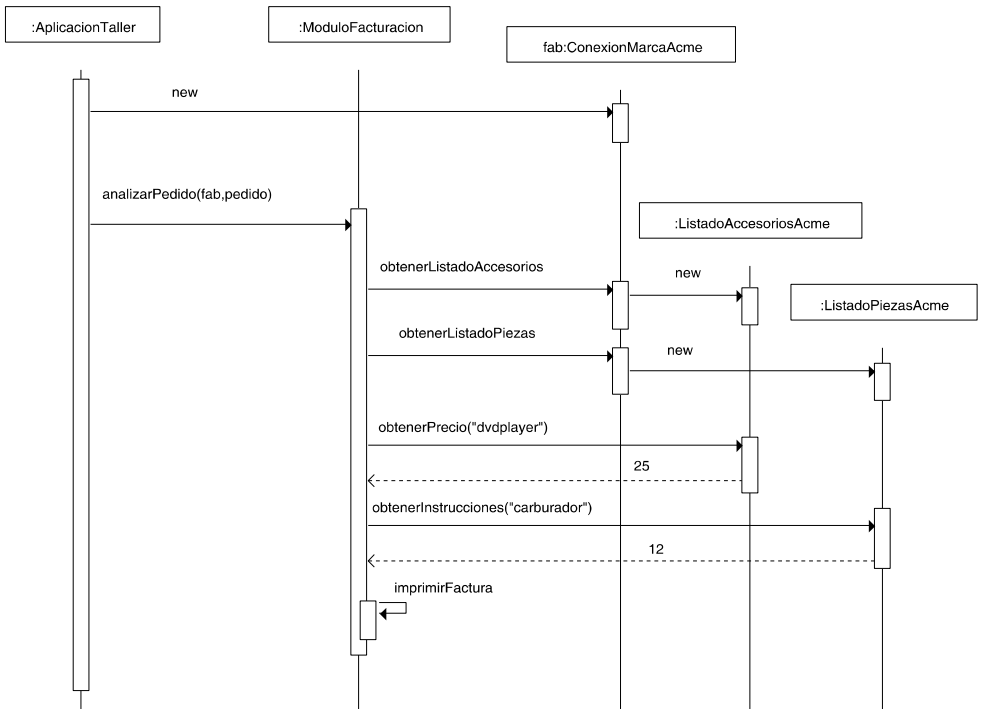


Figura 61.- Diagrama de secuencia que muestra el uso de una fábrica para el manejo polimórfico de diferentes tipos de conexiones.

Existen otros Patrones de Diseño para la creación de objetos. En particular se puede destacar al **patrón Constructor** (***Builder***) y al **patrón Método de Construcción** (***Factory Method***). El patrón Constructor es similar al patrón Fábrica Abstracta salvo en que una Fábrica suele crear colecciones de objetos, mientras que un constructor solo crea un tipo de objetos. Por otro lado, el patrón Método de Construcción se diferencia del patrón Fábrica en que son los propios objetos derivados los que crean, mediante un método de construcción, los objetos concretos que se necesitan en cada momento.

Otra alternativa al uso del patrón Fabrica Abstracta consiste en el uso de inyectores de dependencias. Son objetos que se programan o se configuran para que devuelvan familias concretas de productos cuando cualquier clase de la aplicación le

solicita cierto tipo de producto. Estas técnicas se apoyan en las características de introspección de *Java* y pueden ser complementarias al uso de Fábricas Abstractas.

8.1.2 Adaptador o Envoltorio (*Adapter* o *Wrapper*)

El patrón Envoltorio se utiliza cuando se desea que una clase utilice otra aunque ésta no cumple cierta interfaz obligatoria. Por ejemplo, en el diagrama adjunto la clase `Cliente` solo puede utilizar objetos que cumplan la interfaz `InterfazAmiga`, pero se desea utilizar sobre la clase `Extraña`. Para ello, se crea la clase `Adaptador`, que contiene un objeto de la clase `Extraña` pero que implementa la interfaz `Amiga`. De esta forma, la clase `Usuaria` utiliza, indirectamente, la clase `Extraña`.

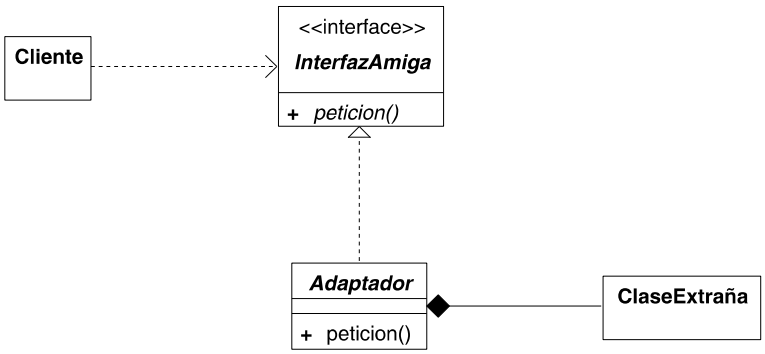


Figura 62.- El cliente accede a la clase extraña utilizando una interfaz que conoce y que envuelve a la clase extraña.

Este patrón se utiliza cuando deseamos almacenar un tipo primitivo de *Java* en un derivado de `Collection`. Por ejemplo, supongamos que deseamos almacenar un `int` en un `Vector`. Como los enteros no son elementos de la clase `Object` es necesario envolverlos en otra clase que contenga el entero y que, al derivar de `Object`, sí pueda ser insertada en el vector. Por eso, y por otras razones, en *Java* existen clases como `Integer`, `Float`, `Boolean` o `Character` que son envoltorios de los tipos primitivos.

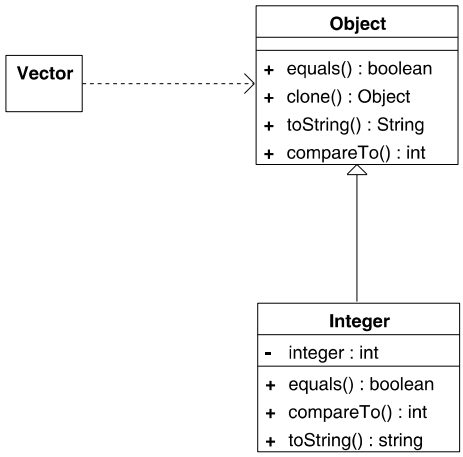


Figura 63.- En este caso el adaptador es `Integer` y adapta al tipo primitivo `int` para que pueda ser usado como un `Object`.

8.1.3 Decorador (*Decorator*)

El patrón Decorador permite añadir nueva funcionalidad a una familia de componentes manteniendo la interfaz del componente. El siguiente diagrama ilustra este patrón. La clase `Componente` es abstracta, la clase `ComponenteConcreto` implementa un comportamiento determinado. La clase `Decorador` contiene un `Componente` en su interior. Cuando se solicita una operación al objeto de la clase `Decorador` esta la deriva al `Componente` que contiene. Las clases derivadas de `Decorador` son los verdaderos Decoradores que implementan una nueva funcionalidad añadida al `Componente` que contienen.

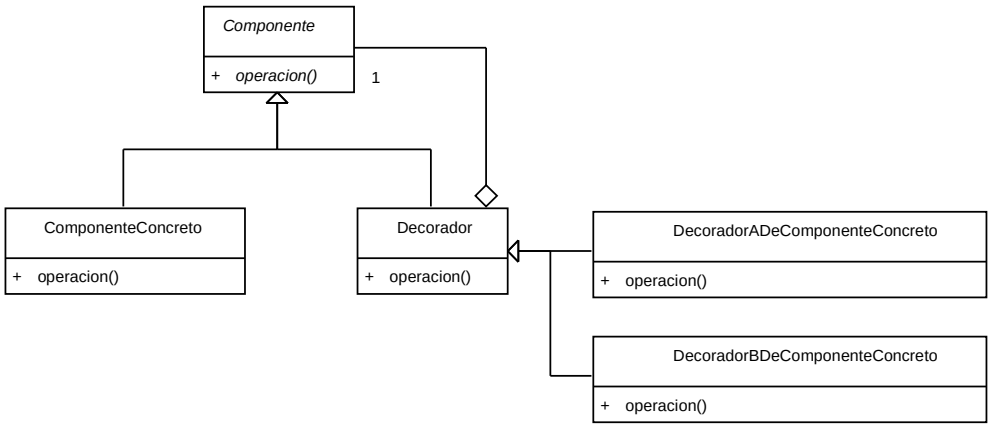


Figura 64.- El decorador `A` añade cierta funcionalidad a cualquier componente manteniendo la interfaz de tal componente.

Por ejemplo, el patrón Decorador se puede utilizar, para añadir cifrado o compresión a las clases de escritura en *Streams*. Así, la clase de la que derivan todas sería `Writer`. Un `Writer` concreto, por ejemplo, es el `FileWriter`. La misión de `WriterDecorator` es la de redirigir las llamadas a los diferentes métodos de la interfaz `Writer` hacia el `Writer` concreto que contiene (`FileWriter`, `PrinterWriter`...). Finalmente, las clases `EncriptWriter` y `ZipWriter` implementan cierta operación sobre el flujo de salida que se dirige hacia el `Writer` concreto contenido en el `Decorador`.

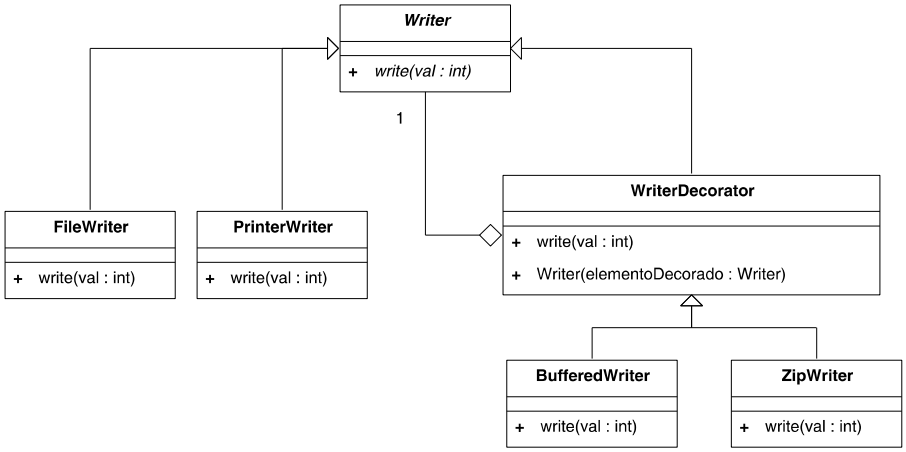


Figura 65.- Decorador para añadir un *buffer* y compresión a un `Writer`.

El siguiente Diagrama de Actividad muestra el comportamiento esperado para este patrón.

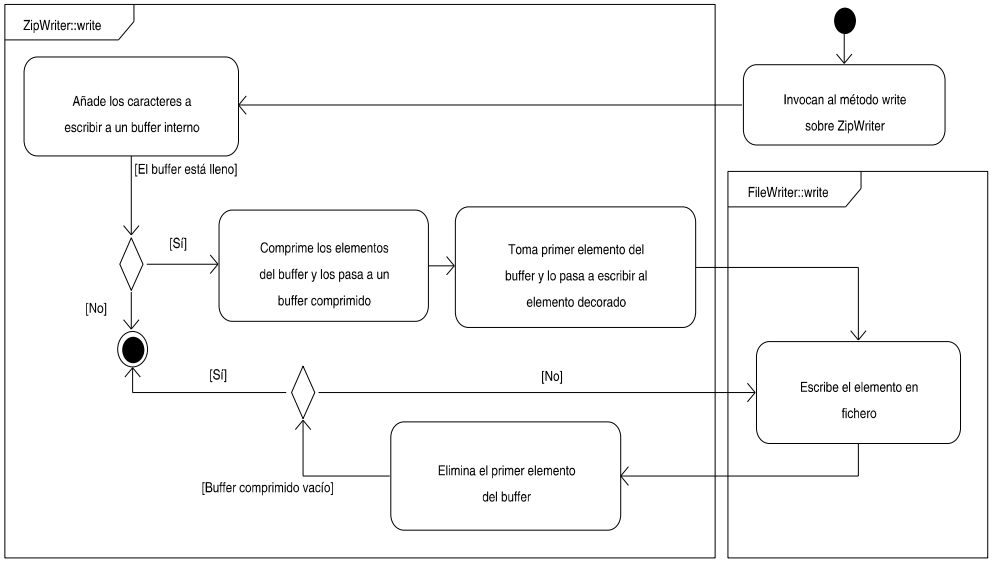


Figura 66.- La escritura pasa primero por el decorador (el `ZipWriter`), el cual decide cuándo enviarla al elemento decorado (el `FileWriter` en este caso).

8.1.4 Composición (*Composite*)

Es muy común la necesidad de crear grafos dirigidos en los que cada nodo puede representar ciertos elementos de un modelo informático. Estos grafos suelen crearse utilizando el patrón Composición. Este patrón se puede definir como jerarquías de objetos que comparten una interfaz y tales que algunos de los objetos pueden formar parte de otros. La siguiente figura describe este patrón. Obsérvese que tanto los objetos de la clase `Elemento` como los de `Compuesto` cumplen la interfaz `Componente`, pero los de clase `Compuesto` además puede contener dentro otros objetos de la clase `Componente`.

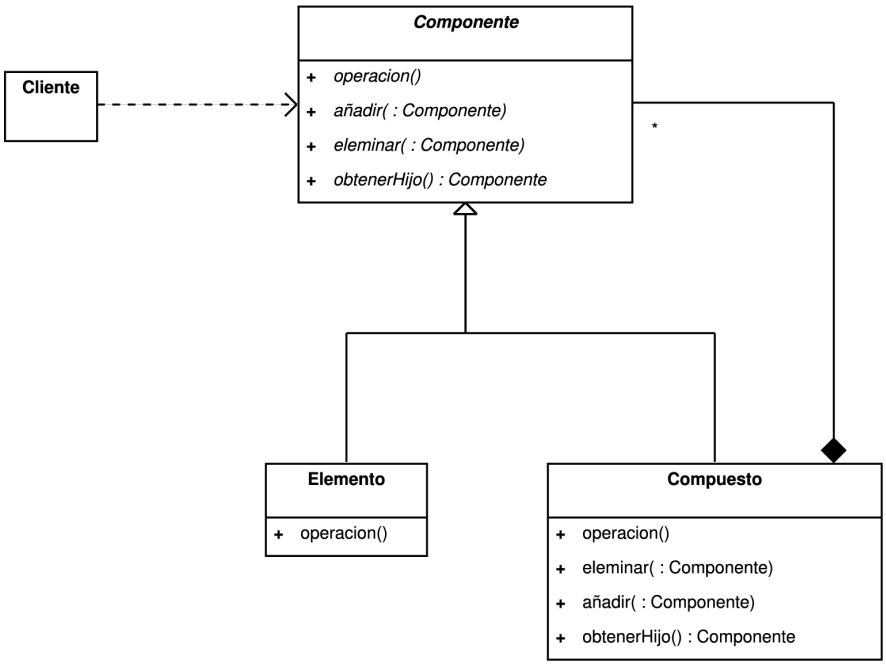


Figura 67.- La clase componente proporciona la interfaz común. Las clases derivadas pueden ser de dos tipos: compuestos (que a su vez agrupan a otros componentes) o simples hojas.

Por ejemplo, en una jerarquía de componentes de dibujo puede ser importante que todos los elementos que se puedan dibujar compartan cierta interfaz, pero además también es importante que unos elementos puedan formar parte de otros (los objetos línea forman parte del objeto cuadrado).

8.1.5 Iterador (*Iterator*)

Habitualmente, cuando se dispone de una colección de objetos se desea recorrerlos. El patrón Iterador permite definir objetos para realizar esta tarea. Un objeto iterador es una especie de apuntador que se puede iniciar apuntando al primer elemento de una colección y que puede desplazarse hasta el último elemento de la misma.

Java dispone de la interfaz `Iterator`, la cual está implementado por las diferentes clases de `Collection`. En realidad, estas clases son fábricas de iteradores que permiten recorrerlas.

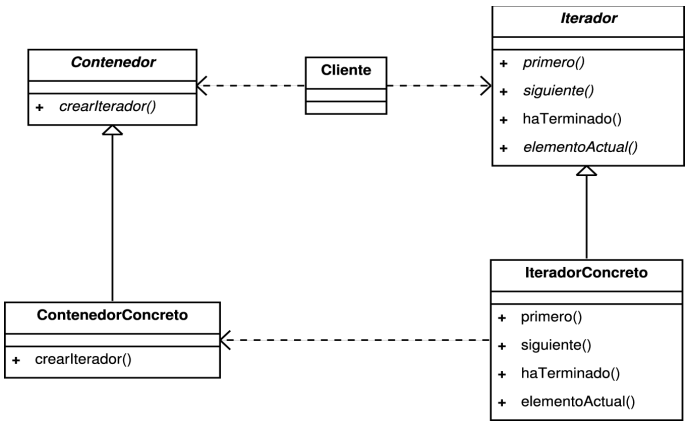


Figura 68.- El iterador concreto permite recorrer el contenedor concreto.

8.1.6 Estrategia (*Strategy*)

En cualquier programa es habitual disponer de un conjunto de algoritmos que comparten alguna propiedad, como que pueden ejecutarse indistintamente sobre unos datos de entrada o que sean de determinado tipo. Ejemplos de tales familias serían las funciones matemáticas (seno, coseno, raíz...) o los filtros gráficos de un programa de dibujo. El patrón Estrategia permite organizar dichas familias de algoritmos, de manera que compartan una interfaz para que luego los clientes de dichas clases puedan utilizarlos indistintamente.

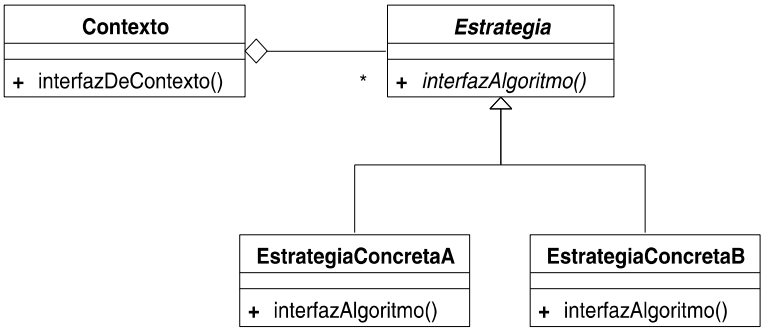


Figura 69.- Los objetos de la clase de contexto que acceden a la familia de algoritmos los hacen de forma abstracta, sin importarles el algoritmo específico que se está utilizando.

Un ejemplo de uso del patrón Estrategia puede ser la implementación de los diferentes algoritmos de ordenación de una lista de números (ver Figura 70). Gracias al patrón Estrategia el usuario del contexto puede modificar su criterio de ordenación de forma dinámica.

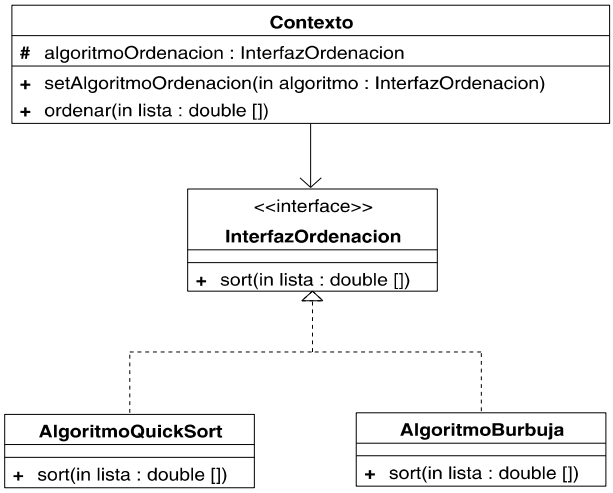


Figura 70.- Un usuario puede utilizar cualquiera de los algoritmos que cumplan la interfaz de ordenación a través del objeto `Contexto`.

8.1.7 Comando (*Command*)

Este patrón encapsula las operaciones que realiza un objeto de forma que éstas sean a su vez objetos que cumplen una misma interfaz. Esto permite realizar, de manera sencilla, tareas como: agrupar o encolar operaciones, deshacer operaciones y parametrizar otros objetos con dichas operaciones de forma sencilla. Además, fomenta que añadir nuevos comandos sea una tarea simple y aislada. El Diagrama Estático de Clases adjunto describe su estructura estática y el Diagrama de Secuencia de la Figura 72 explica su comportamiento dinámico.

El patrón Comando se podría utilizar, por ejemplo, para ordenar los comandos que se pueden ejecutar desde un intérprete de consola (ver Figura 73). Si el interprete utiliza los comandos solo a través de la interfaz común, sin conocer en cada momento el comando concreto que se está ejecutando, una de las ventajas que se obtienen consiste en que el número de comandos puede crecer sin modificar dicho interprete.

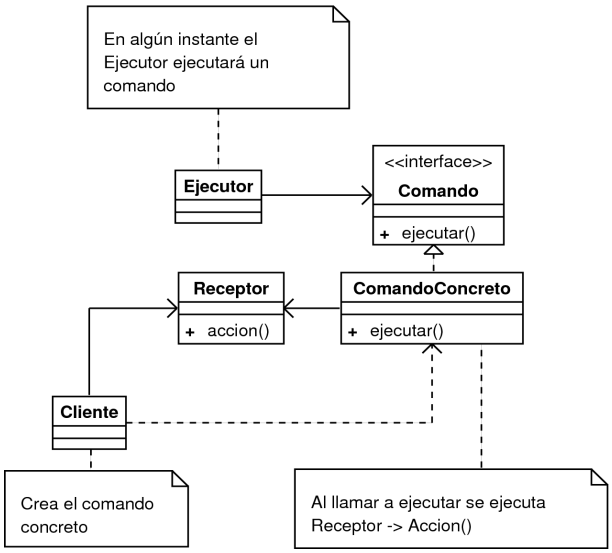


Figura 71.- Diagrama de clases del patrón Comando.

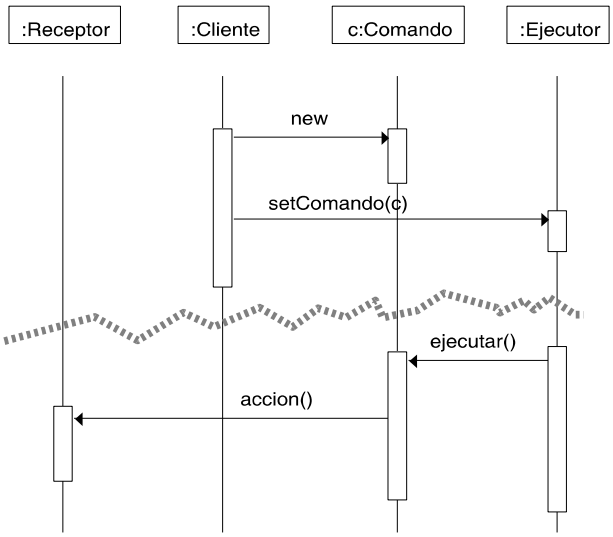


Figura 72.- El cliente crea los comandos y los asocia al ejecutor. Más tarde, ejecuta el comando que corresponda. Dicho comando, que conoce al receptor de su orden, ejecuta la acción asociada al receptor que conoce .

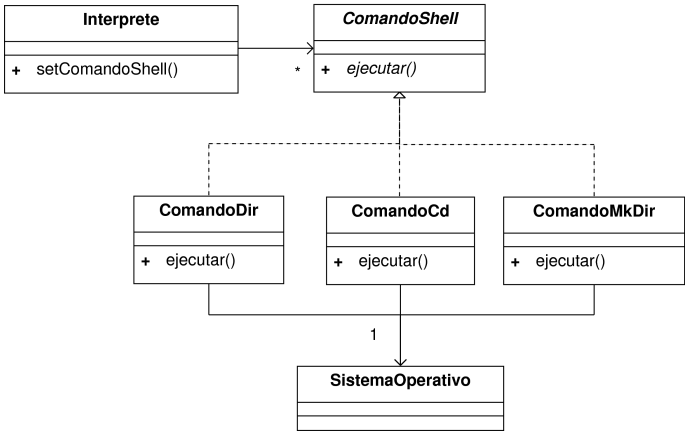


Figura 73.- El objeto de clase Interprete sólo necesita conocer la interfaz de la clase `ComandoShell`. En este caso, el sistema operativo sería el receptor y el interprete sería el ejecutor.

8.1.8 Observador (*Observer*)

Este patrón crea una relación entre objetos en la que uno de ellos es observado por el resto, de manera que cuando el objeto observado cambia el resto puede automáticamente realiza alguna acción. En la Figura 74 se presenta el Diagrama Estático de Clases de este patrón. Dinámicamente, cuando cambia el estado del objeto `ElementoObservadoConcreto` se ejecuta el método `avisar()`, el cual llama al método `actuar()` de cada observador.

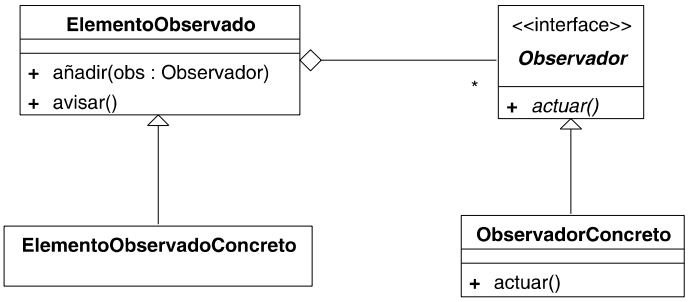


Figura 74.- El Elemento observado avisa a los observadores cuando su estado cambia.

El patrón Observador lo utiliza *Java* para implementar el código que se ejecuta cuando un objeto de tipo componente cambia de estado. *Java* crea un objeto de clase `Listener` (escuchador en vez de observador) para cada operación que se realiza con un botón (elemento observado en este caso). Ese objeto `Listener` contiene el código que se ejecuta al realizar la operación sobre el botón. Así, cuando un usuario pulsa un botón y el estado del componente botón cambia el `Listener` que lo observa es capaz de ejecutar cierto código.

8.2. Algunos principios útiles en POO

En este apartado se presentan algunos principios que se deben seguir al realizar el diseño de un programa orientado a objetos.

8.2.1 El principio abierto-cerrado

Bertrand Meyer estableció el siguiente principio en 1998:

Todas las entidades de software (clases, módulos, funciones...) deben estar abiertas para extensiones y cerradas para modificaciones.

Este principio intenta evitar que un cambio en un programa produzca una cascada de cambios en módulos dependientes. Los programas que se hagan siguiendo este principio deberán tener vías para que cuando los requisitos cambien podamos extender el comportamiento añadiendo nuevo código (abierto a extensión), pero sin cambiar el código que ya funciona (cerrado a modificación).

La mejor forma de conseguir que un código siga este principio consiste en fomentar el uso de clases abstractas o interfaces en las relaciones entre objetos.

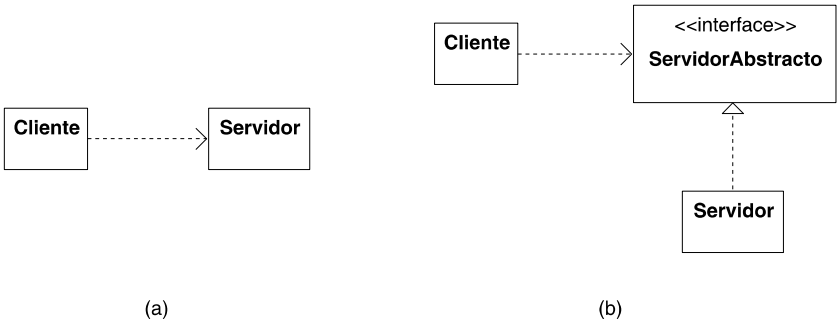


Figura 75.- En una relación entre clases debe preferirse siempre el esquema (b) al (a).

Como consecuencia del principio abierto-cerrado aparecen dos corolarios:

- Todas las propiedades deben ser privadas.- Si una propiedad no es privada ningún módulo externo que dependa de esa variable podrá estar cerrada a modificaciones cuando por ejemplo se cambie el comportamiento de la propiedad o se elimine. Además cualquier modulo externo que utilice la variable de una forma no esperada por el resto de módulos clientes de la variable hará que éstos fallen.

- Evitar la identificación dinámica (uso del *casting* de referencias y de `instanceof`). Si un módulo necesita identificar el tipo de un objeto en tiempo de ejecución para tomar una decisión, este módulo podría no estar cerrado a modificaciones que ocurrirían cuando apareciesen nuevas clases de objetos.

8.2.2 El principio de Liskov

Barbara Liskov estableció el siguiente principio:

Toda función que utiliza referencias a un objeto de una clase A debe ser capaz de usar objetos de clases derivadas de A sin saberlo.

Este principio ofrece un camino para que el código sea más fácil de extender, ya que fomenta la creación de métodos que pueden funcionar con clases que aún no han sido creadas.

8.2.3 El principio de segregación de interfaces

Robert C. Martin estableció el siguiente principio:

Es preferible varias interfaces específicas que una única interfaz general.

Seguir este principio minimiza el impacto del cambio de una interfaz, ya que no existen grandes interfaces de las que dependen multitud módulos.

8.2.4 Maximizar el uso de clases inmutables

Las clases inmutables son aquellas cuyos objetos no cambian de estado una vez creados. Por ejemplo, `String` es una clase inmutable. Con objeto de forzar este comportamiento todas las propiedades de la clase deben ser finales. La inmutabilidad simplifica enormemente la implementación de una clase. Además hace posible el uso de un mismo objeto desde diferentes *threads* simultáneamente sin usar mecanismos de exclusión. Evidentemente, es imposible que ciertas clases, las que mantienen el estado de un programa, sean inmutables.

8.2.5 Preferir composición frente a herencia

Ya se ha comentado que la herencia rompe la encapsulación. Tal y como se ha comprobado con el patrón Decorador, siempre es posible obtener un comportamiento similar a la herencia utilizando composición. Para forzar esta preferencia es una buena práctica siempre definir las clases y los métodos como finales, evitando de esta forma que se pueda heredar de ellos. Solo en los casos en los que la herencia se planifique, como en el patrón Estrategia, se debe omitir el modificador `final`.

8.2.6 Principio de responsabilidad única

Una clase debe tener una única responsabilidad y dicha responsabilidad debe estar completamente encapsulada en dicha clase. Robert C. Martin define este principio como sigue:

Una clase debe tener un único motivo para cambiar de estado.

8.2.7 Eliminar duplicaciones

Andy Hunt denominó a este principio DRY (*Don't repeat your self*). Siempre que existe una duplicación se debe a que no se ha abstraído lo suficiente. Probablemente, la duplicación se pueda eliminar con la introducción de un nuevo método o una nueva clase.

La mayoría de los patrones que han aparecido en los últimos 15 años han sido creados para eliminar las duplicaciones en el código.

8.2.8 Principio de inversión de dependencia

Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.

8.2.9 Preferir el polimorfismo a los bloques `if/else` o `switch/case`

Normalmente, el uso de uno de estos bloques indica que la clase está rompiendo el principio de responsabilidad única. Si en una clase tenemos un bloque `if/else`, esa clase unas veces tendrá una responsabilidad (cuando entre por la rama `if`) y otras veces tendrá otra diferente (cuando entre por la rama `else`). Prácticamente la totalidad de los bloques `if/else` y de los bloques `switch/case` pueden reemplazarse mediante polimorfismo.

8.2.10 Conclusiones

El descubrimiento de estructuras comunes (patrones) en diferentes tipos de programas, puede ayudar a identificar buenas estructuras que se pueden reutilizar en diferentes implementaciones. Es por ello que, desde aquí, se anima al lector a que profundice en los conocimientos de Programación Orientada a Objetos que acaba de adquirir mediante este camino.

8.3. Lecturas recomendadas

"Patrones de Diseño", Erich Gamma y otros, Addison Wesley, 2002. Este libro es la principal guía de referencia en cuanto a patrones de diseño.

"Piensa en *Java*. Eckel", 2ª Edición, Addison Wesley, 2002. Su anexo C contiene un gran compendio de principios y buenas prácticas.

"*Clean Code*", Robert C. Martin.

"*Design Principles and Design Patterns*", Robert C. Martin, www.objectmentor.com, 2000.

8.4. Ejercicios

Ejercicio 1

Realice el diseño de un programa que implemente una Hoja de Cálculo.

El programa contendrá en cada momento un libro abierto el cual se compondrá de varias hojas. A su vez, cada hoja contiene celdas en cada una de las cuales se pueden disponer texto o fórmulas.

El texto consistirá en un conjunto de caracteres alfanuméricos entre comas simples. Las fórmulas se compondrán de constantes numéricas (2, 3.14...), operadores simples (+-*/) y funciones predefinidas (cos, sen, tg ...). El anexo 1 describe la gramática que deberán seguir las fórmulas.

Las celdas se disponen en forma de retícula, formada por filas y columnas. Cada columna y cada fila estará etiquetada por un número consecutivo. Podrá haber un número indeterminado de filas y columnas, aunque en pantalla sólo se podrán ver simultáneamente un subconjunto del total. En todo momento habrá siempre una hoja activa, de manera que las operaciones con las celdas se referirán siempre a una hoja concreta.

La interacción con el programa, que podrá ser en modo texto, consistirá en la escritura de instrucciones mediante el teclado. Así, el sistema ofrecerá un *prompt* y el usuario podrá escribir una instrucción, la cual se ejecutará tras presionar la tecla *Return*. El anexo 2 describe el juego de instrucciones que deberá admitir el programa.

Fórmulas

Las fórmulas que se requiere que contengan las celdas son las generadas por la siguiente gramática en notación BNF:

```
<FÓRMULA> ::=  <FÓRMULA> <OPERADOR> <FÓRMULA> | <REAL> | <FUNCIÓN>(<FORMULA>) |    <REF>

<OPERADOR>::=  + | - | * | /

<FUNCIÓN> ::=  sen | cos | tg

<REF> ::=   celda(<NÚMERO,NÚMERO>)

<REAL> ::= <NÚMERO>.<NÚMERO> | <NÚMERO>

<NÚMERO> ::=   <DÍGITO> | <DÍGITO> <NÚMERO>

<DÍGITO> ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Donde:

cos(<fórmula>).- Calcula el coseno de una fórmula.

sen(<fórmula>).- Calcula el seno de una fórmula.

tg(<fórmula>).- Calcula la tangente de una fórmula.

celda(<fila,columna>).- Devuelve el valor contenido en la celda .

Juego de instrucciones

Las órdenes que se desea que tenga la hoja de cálculo está compuesto por las siguientes instrucciones:

CrearLibro.- Esta operación elimina las hojas de cálculo actualmente abiertas y crea 3 hojas vacías etiquetadas como hoja1, hoja2 y hoja3. Además se pondrá la hoja1 como hoja activa.

CrearHoja <nombre>.- Crea una hoja dentro el libro actual y la dispone como hoja actual. Los nombres de las hojas no podrán contener espacios, además no se podrá crear una hoja si tiene el mismo nombre que otra que ya exista.

BorrarHoja <nombre>.- Borra una hoja dentro el libro actual. Si era la hoja activa dispone otra como activa. Si no quedan mas hojas automáticamente crea una nueva.

Renombrar <nombre hoja><nuevo nombre>.- Esta operación renombra una hoja. Los nombres de las hojas no podrán contener espacios, además no se podrá crear una hoja si tiene el mismo nombre que otra que ya exista.

HojaActual <nombre>.- Permite seleccionar cual es la hoja actual.

Celda(<fila>,<columna>)=[<formula>|<'texto'>].- Permite asignar un valor nuevo a una celda. Si el valor es de tipo alfanumérico deberá ir entre comillas simples, en otro caso corresponderá a una fórmula que seguirá la gramática del anexo 1.

Así, para asignar un valor a una celda se utilizará la siguiente notación:

```
Celda(<fila>,<columna>)='texto'
Celda(<fila>,<columna>)=fórmula
```

Por ejemplo:

```
Celda(1,1)='Radio'
Celda(1,2)=25
Celda(2,1)='Perímetro'
Celda(2,2)=Celda(1,2)*3.14*2;
```

Mostrar(<fila>, <columna>).- Muestra por pantalla el contenido de las celdas correspondientes a 7 columnas y 10 filas a partir de la casilla indicada por el par (<fila>, <columna>).

Cada columna y cada fila estará encabezada por un número consecutivo que la identifica. Además, todo el texto estará encabezado por el nombre de la hoja.

Se utilizará exactamente 9 caracteres para presentar el contenido de cada celda y un espacio como separación con la celda siguiente. Con el fin de que aparezca ordenado el contenido de cada celda se completará con espacios cuando el tamaño sea menor y se recortará el contenido por la derecha cuando el tamaño sea mayor.

Si una celda contiene texto alfanumérico se presenta tal texto en el lugar asignado.

Si una celda contiene una fórmula se presentará, en el lugar asignado, el resultado de la aplicación de la fórmula en ese momento y no el texto de la fórmula.

Por ejemplo, `Mostrar(0,0)` muestra desde la celda (0,0) hasta la celda (6,9) presentaría:

```
Hoja: Hoja1

          0|         1|         2|         3|         4|  ...
0
1                Radio         25
2            Perímetro        157
3
...
```

Deberá detectarse la aparición de referencias circulares, por ejemplo almacenando si una celda ya ha sido calculada, para impedir que vuelva a intentarse su cálculo.

Se devolverá el texto `ERROR` si una fórmula resulta en un error, por ejemplo si se detecta referencia circular al calcular el valor de una celda.

Cargar.-<"ruta nombre de fichero">.- Permite cargar un libro de disco.

Salvar <"ruta y nombre de fichero">.- Permite guardar un libro con todas sus hojas y el contenido de sus celdas a disco.

Anexo A Solución de los ejercicios

A.1 Ejercicios del capítulo 1

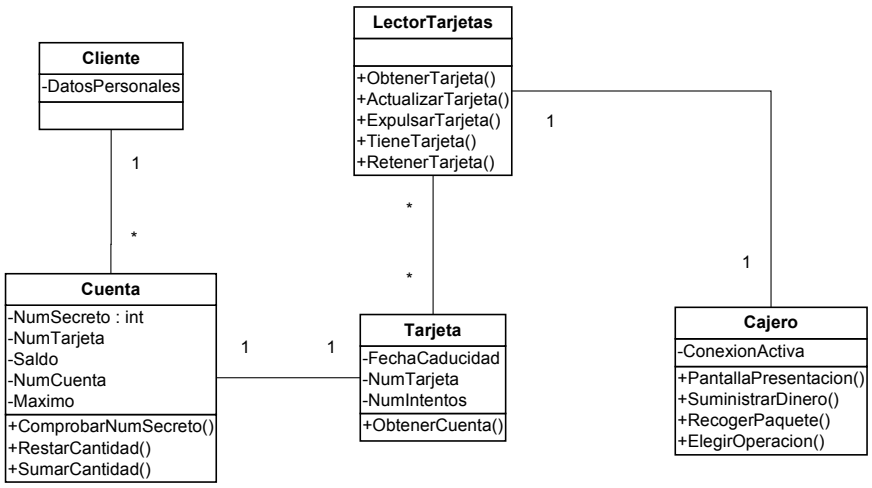
A.1.1 Solución del ejercicio 1

En este caso se propone una solución que utiliza seis clases: tarjeta, cuenta, cliente, lector de tarjetas, cajero y operación. Las siguientes líneas describen a grandes rasgos las responsabilidades de tales objetos:

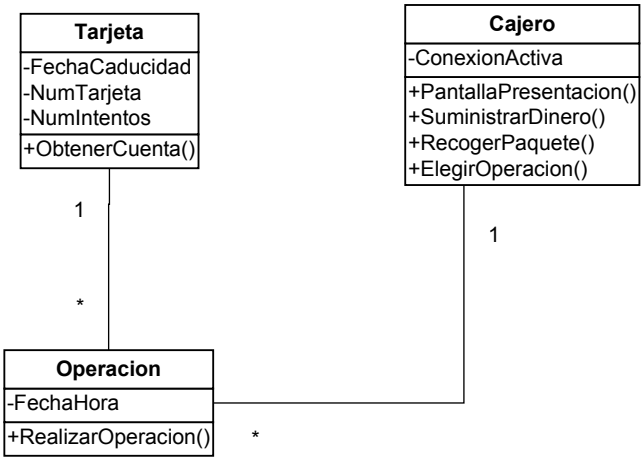
- `Tarjeta`.- Abstracción que se encarga de gestionar los datos relativos a la tarjeta física.
- `LectorTarjetas`.- Abstracción que se encarga de construir objetos tarjeta a partir de las tarjetas reales. Encapsula el manejo del *hardware* específico de lectura y escritura de tarjetas.
- `Cajero`.- Abstracción que se encarga de coordinar la realización de las operaciones: verificar introducción de tarjeta, solicitud y comprobación del número clave, selección de la operación y devolución o retención de la tarjeta.
- `Cuenta`.- Abstracción que contiene la información relativa a la cuenta de un cliente y que se encarga de autorizar o denegar las operaciones solicitadas. Esta abstracción encapsula el acceso a la base de datos del banco que contiene los datos del cliente.
- `Cliente`.- Abstracción que mantiene los datos personales del cliente.
- `Operación`.- Abstracción que realiza las operaciones que se solicitan.

A continuación se presentan varios Diagramas Estáticos de Clases de *UML*. Estos diagramas utilizan los elementos que ya se han presentado. Es importante notar que no suele ser conveniente representar todas las clases en un mismo diagrama, debido a que los diagramas deben ser simples para que realmente sean utilizables al describir un problema. Es preferible tener varios diagramas pequeños, cada uno centrándose en una perspectiva concreta del diseño, que un único diagrama complejo.

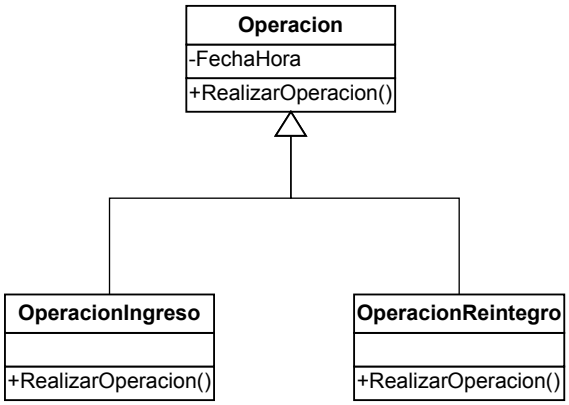
El primer diagrama muestra la conexión entre las principales clases involucradas en el proceso.



En el segundo diagrama se presenta una vista más parcial, en la que se profundiza en las operaciones que se pueden realizar con una tarjeta.



En este punto se decide introducir en el objeto `Operación` las operaciones que se pueden realizar: reintegro e ingreso. Al hacerlo resulta que la clase operación se podría complicar excesivamente. Además, si se quisiera ampliar el número de operaciones que se pueden realizar debería cambiarse la clase operación para añadirle nuevos métodos. Por ello se opta por usar herencia de la clase `Operación` para implementar cada una de las operaciones. Esto da lugar al diagrama de la figura siguiente. En él se aprecia que cada clase derivada de `Operación` deberá implementar el método `realizarOperacion()` de manera conveniente.



Es importante notar que la descomposición en objetos que se realiza depende del analista. Diferentes analistas pueden tener visiones diferentes del problema y ser todas correctas.

A.2 Ejercicios del capítulo 2

A.2.1 Solución del ejercicio 1

Una propiedad de clase es aquella que se declara usando `static` y que por ello comparte el mismo valor para todos los objetos de la misma clase. La propiedad de instancia tiene un valor individual para cada objeto de una clase.

A.2.2 Solución del ejercicio 2

En *Java* existe un proceso, llamado "recolector de basura", que se encarga de realizar esta tarea de manera automática.

A.2.3 Solución del ejercicio 3

En la línea 7 se utiliza = cuando debería usarse = =.

En la línea 8 se accede al método `value` de `aux2` cuando esa referencia no apunta a ningún objeto.

En la línea 10 se realiza una operación que nunca se efectuará porque está detrás de `return`.

A.3 Ejercicios del capítulo 3

A.3.1 Solución del ejercicio 1

Sí, porque al interfaz de una clase debe mantener su visibilidad a través de la herencia para que el polimorfismo sea posible.

A.3.2 Solución del ejercicio 2

Básicamente, tal clase tiene la misma utilidad que una interfaz. Estas clases, que suelen llamarse abstractas puras, suelen usarse como clase base de una jerarquía de objetos para la que se desea que comparta la interfaz definida por la clase.

A.3.3 Solución del ejercicio 3

```
interface ElementoOrdenable {
   public boolean mayorQue(ElementoOrdenable e);
}


class ArbolBinario {
   private ElementoOrdenable e_raiz;
   private ArbolBinario      arbol_derecha;
   private ArbolBinario      arbol_izquierda;

   public ArbolBinario izquierda() {
      return arbol_izquierda;
   }

   public ArbolBinario derecha() {
      return arbol_derecha;
   }

   public ElementoOrdenable obtenerElemetoRaiz() {
      return e_raiz;
   }

   public void Insertar (ElementoOrdenable e) {
      if (e_raiz == null)
         e_raiz = e;
      else {
         if (e.MayorQue(e_raiz)) {
            if (arbol_derecha == null)
                     arbol_derecha = new ArbolBinario();
            arbol_derecha.Insertar(e);
         }
         else {
            if (arbol_izquierda == null)
                     arbol_izquierda = new ArbolBinario();
            arbol_izquierda.Insertar(e);
         }
      }
   }
}


class EnteroOrdenable implements ElementoOrdenable {
   public int elemento;
   public EnteroOrdenable(int e) {
      elemento = e;
   }

   public boolean mayorQue(ElementoOrdenable e) {
      EnteroOrdenable aux = (EnteroOrdenable) e;
      return (elemento > aux.elemento);
   }
}


class StringOrdenable implements ElementoOrdenable {
   public String elemento;
   public StringOrdenable(String e) {
      elemento = e;
   }
   public boolean mayorQue(ElementoOrdenable e) {
      StringOrdenable aux = (StringOrdenable) e;
      return (elemento.length() > aux.elemento.length());
   }
}

class Solucion1 {
   public void main(String args[]) {
      EnteroOrdenable a = new EnteroOrdenable(3);
      EnteroOrdenable b = new EnteroOrdenable(5);
      EnteroOrdenable c = new EnteroOrdenable(11);
      EnteroOrdenable d = new EnteroOrdenable(7);
      EnteroOrdenable e = new EnteroOrdenable(13);

      ArbolBinario arbol = new ArbolBinario();
      arbol.Insertar(a);
      arbol.Insertar(b);
      arbol.Insertar(c);
      arbol.Insertar(d);
      arbol.Insertar(e);
   }
}
```

A.4 Ejercicios del capítulo 4

A.4.1 Solución del ejercicio 1

a) La lista se implementa utilizando nodos doblemente enlazados por razones de eficiencia, aunque podría haberse implementado con nodos con un enlace simple.

```
class Lista {

  class Nodo{
    Object valor;
    Nodo siguiente;
    Nodo anterior;
  }

  private Nodo primero;
  private Nodo ultimo;
  private int num_nodos = 0;

  public void push_back(Object o) {
    Nodo nuevo = new Nodo();
    nuevo.valor = o;
    nuevo.anterior = ultimo;
    if (ultimo != null)
      ultimo.siguiente = nuevo;
    else
      primero = nuevo;
    ultimo = nuevo;
    num_nodos++;
  }

  public void push_front(Object o) {
    Nodo nuevo = new Nodo();
    nuevo.valor = o;
    nuevo.siguiente = primero;
    if (primero != null)
      primero.anterior = nuevo;
    else
      ultimo = nuevo;
    primero = nuevo;
    num_nodos++;
  }

  public void pop_back(){
    if (ultimo != null){
      ultimo = ultimo.anterior;
      num_nodos--;
    }
  }

  public void pop_front(){
    if (primero != null){
      primero = primero.siguiente;
      num_nodos--;
    }
  }

  public Object front(){
    if (primero != null)
      return primero.valor;
  }

  public Object back(){
    if (ultimo != null)
      return ultimo.valor;
  }

  public int size(){
    return num_nodos;
  }

  public void clear(){
    primero = null;
    ultimo = null;
    num_nodos = 0;
  }
}
```

b) Al heredar sólo habría que añadir los cuatro métodos relacionados con los tipos de los valores contenidos. El resto se hereda.

```
class ListaEnteros1 extends Lista {

  public void push_back_int (Integer i) {
    super.push_back(i);
```

```
  }

  public void push_front_int (Integer i) {
    super.push_front(i);
  }

  public Integer front_int () {
    return (Integer) super.front();
  }

  public Integer back_int() {
    return (Integer) super.back();
  }
}
```

c) Usando composición se declara una clase que contiene una lista y se definen métodos similares a los de la Lista. Estos métodos serán idénticos exceptuando a los relacionados con los tipos de los valores contenidos que estarán especializados en `Integer`.

```
class ListaEnteros2 {
  private Lista lista = new Lista();

  public void pushBack(Integer i) {
    lista.push_back(i);
  }

  public void pushFront(Integer i) {
    lista.push_front(i);
  }

  public Integer front() {
    return (Integer) lista.front();
  }

  public Integer back() {
    return (Integer) lista.back();
  }

  public void popBBack(){
    lista.pop_back();
  }

  public void popFront(){
    lista.pop_front();
  }

  public void clear(){
    lista.clear();
  }
  public int size(){
    return lista.size();
  }
}
```

d) El objetivo es crear una lista en la que sólo se puedan introducir y extraer objetos de la clase `Integer`. En `ListaEnteros1` se heredan todos los métodos y en especial aquellos que permiten insertar objetos derivados de `Object` (es decir cualquier tipo de objeto). Sin embargo `ListaEnteros2` permite especificar una interfaz que sólo deja tratar `Integer`. Por eso en este caso es mejor usar composición y, por tanto, es mejor el enfoque de `ListaEnteros2`.

Se debe usar la clase `Integer` porque `int` es un tipo primitivo (no es una clase) y eso implica que un valor `int` no se puede utilizar desde una referencia a objetos de clase `Object`.

A.5 Ejercicios del capítulo 5

A.5.1 Solución del ejercicio 1

Los Diagramas de Actividad son diagramas de estados que permiten representar las relaciones entre clases y que son especialmente útiles cuando existe herencia entre las clases que se relacionan. Mientras, los Diagramas de Secuencia son Diagramas de Interacción y permiten representar claramente las situaciones en las que se relacionan muchos objetos de diferentes clases. Así, aunque ambos diagramas se utilizan para representar el comportamiento dinámico de los objetos de una o más clases, no podemos decir que sean equivalentes sino complementarios.

A.5.2 Solución del ejercicio 2

A continuación se propone una solución al problema, aunque hay que decir que existen otras muchas diferentes igualmente válidas.

a) Se proponen las siguientes clases:

`TipoBillete`.- Encargada de contener y gestionar la información de los diferentes tipos de billetes que maneja en un momento dado el expendedor.

`ListaTipoBilletes`.- Encargada de contener la lista de tipos de billetes que puede dar en un momento dado el expendedor.

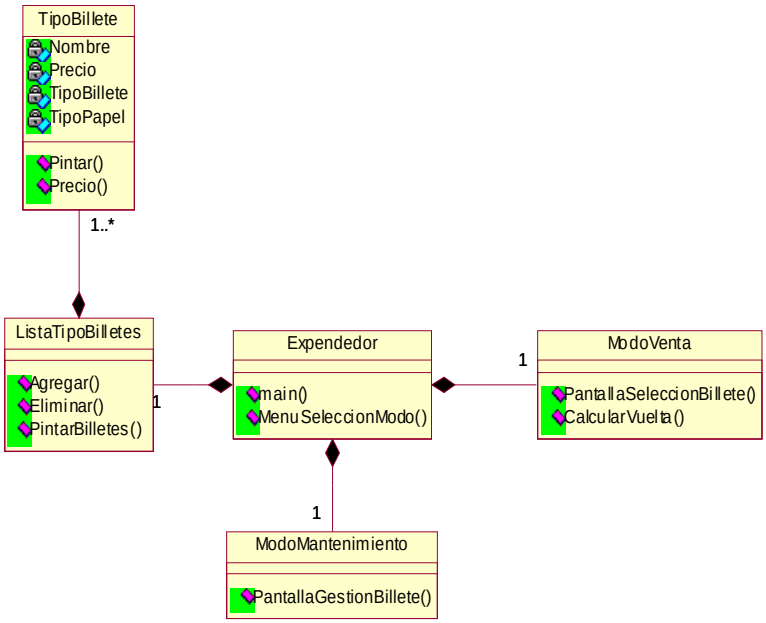
`Expendedor`.- Clase en la que se encuentra el método de arranque (`main`) y que se encarga de permitir alternar entre los diferentes modos operativos.

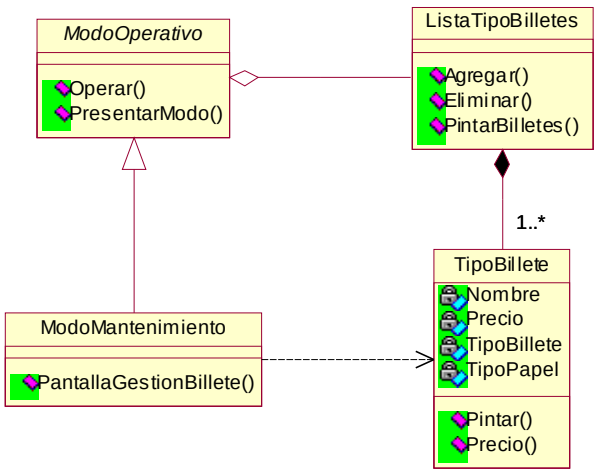
`ModoMantenimiento`.- Clase que gestiona el modo de operación de mantenimiento (pantallas y secuencia de operaciones).

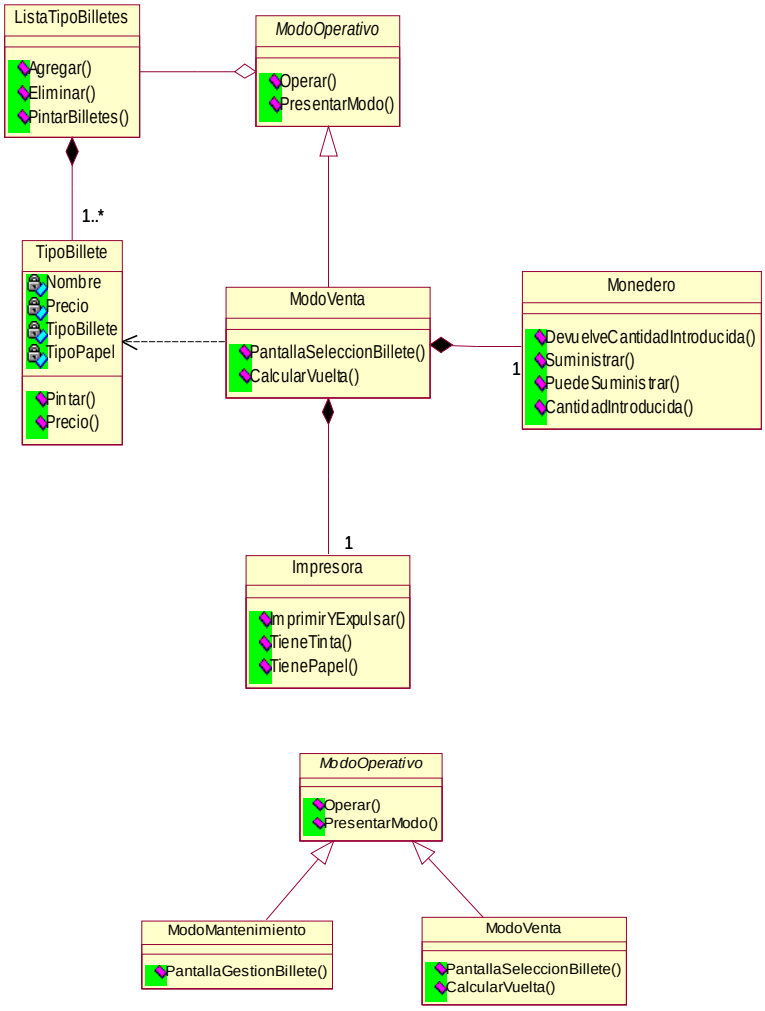
`ModoVenta`.- Clase que gestiona el modo de operación de venta (pantallas y secuencia de operaciones).

`ModoOperativo`.- Clase abstracta de la que heredan la clase `ModoMantenimiento` y la clase `ModoVenta`. Esta clase facilita la futura introducción del *software* correspondiente a nuevos modos que puedan aparecer gracias a que dota a todos los modos de una interfaz común.

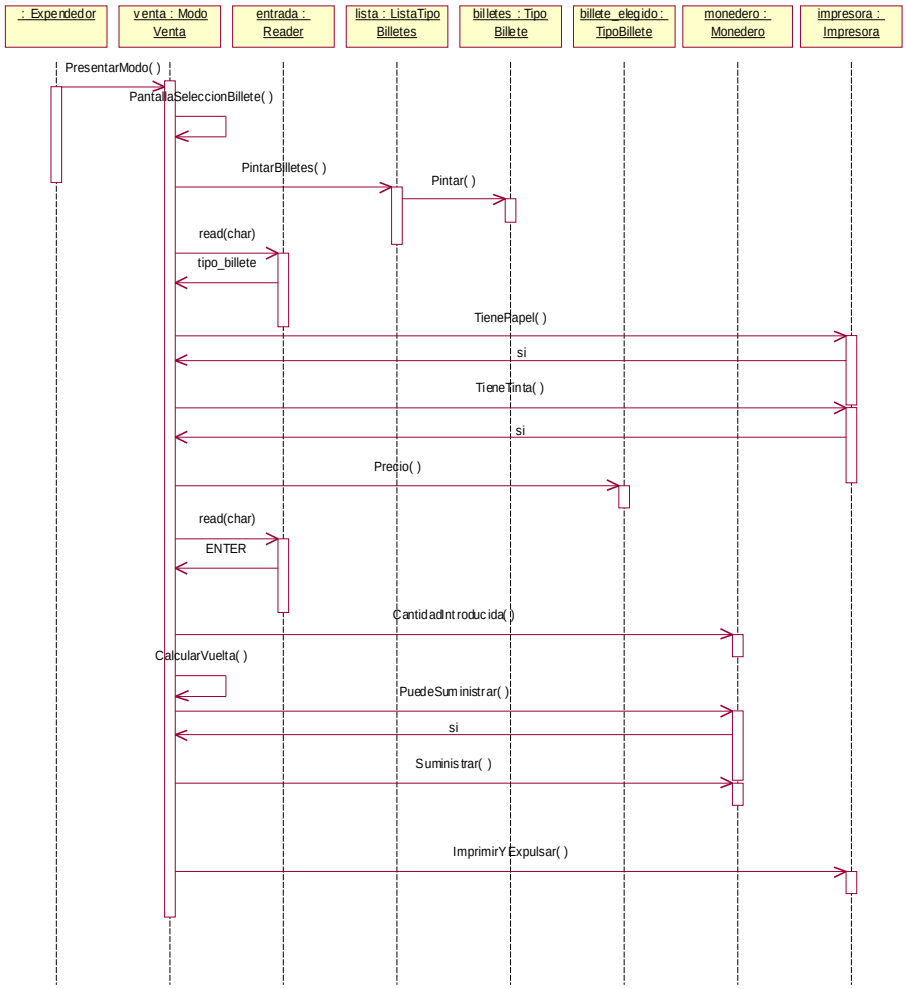
b) Los siguientes diagramas ilustran diferentes perspectivas estáticas de las clases expuestas en el apartado anterior.





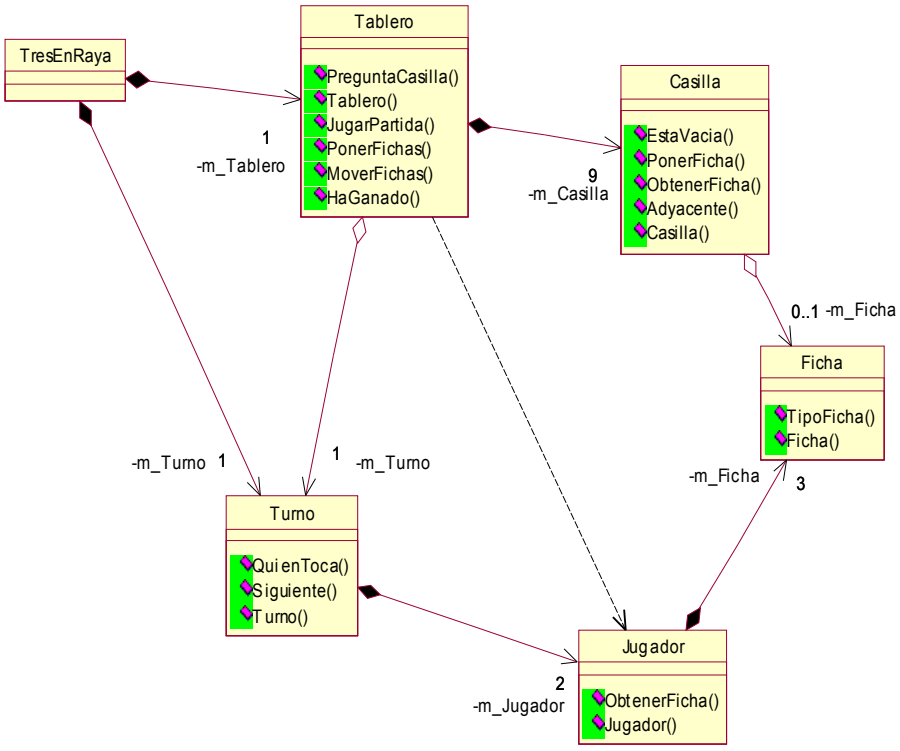

ListaTipoBilletes
Agregar()
Eliminar()
PintarBilletes()
ModoOperativo
Operar()
PresentarModo()
1..*
TipoBillete
Nombre
Precio
TipoBillete
TipoPapel
Pintar()
Precio()
ModoVenta
PantallaSeleccionBillete()
CalcularVuelta()
Monedero
DevuelveCantidadIntroducida()
Suministrar()
PuedeSuministrar()
CantidadIntroducida()
1
1
Impresora
ImprimirYExpulsar()
TieneTinta()
TienePapel()
ModoOperativo
Operar()
PresentarModo()
ModoMantenimiento
PantallaGestionBillete()
ModoVenta
PantallaSeleccionBillete()
CalcularVuelta()

c) El Diagrama de Secuencia para una operación válida con devolución de cambio resulta:



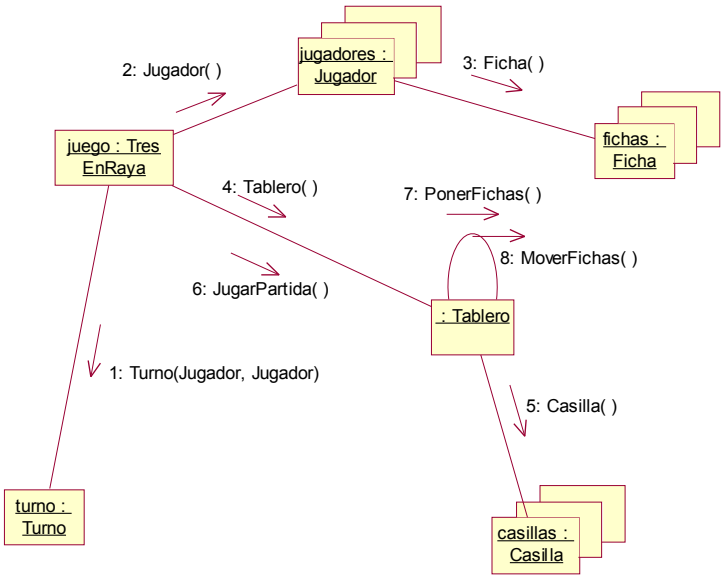
A.5.3 Solución del ejercicio 3

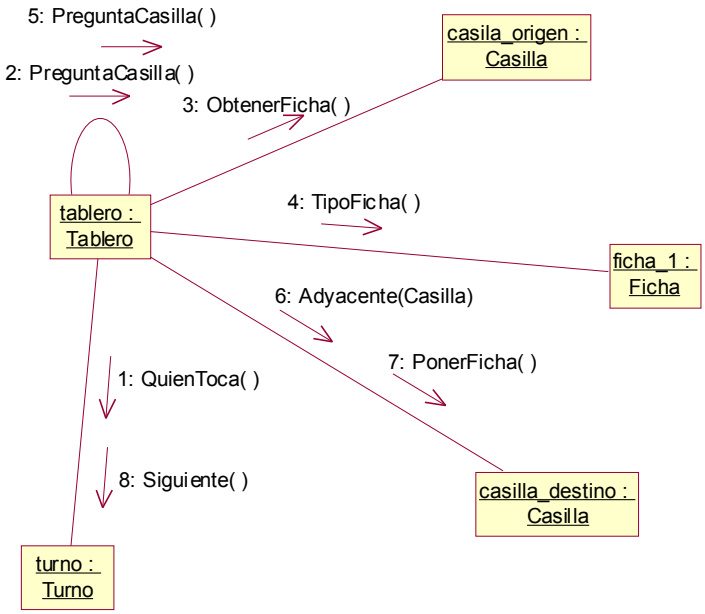
Primeramente se propone el Diagrama Estático de Clases de la figura siguiente. En este diagrama se aprecia la clase `Tablero` (responsable de contener las fichas y verificar la legalidad de los movimiento), la clase `Casilla` (responsable de contener una o ninguna ficha y comprobar su adyacencia a otras casillas), la clase `Ficha` (responsable de conocer a que jugador pertenece) y la clase `Jugador` (responsable de interactuar con la persona que juega). También aparece la clase `TresEnRaya`, en la que residirá la función estática `main()` que iniciará el programa. Por último aparece una clase que puede sorprender: la clase `Turno`. Esta clase se usará para controlar la alternancia entre los jugadores. Es interesante esta clase, porque las anteriores son muy evidentes, siendo meras representadoras de la situación estática del juego. Por el contrario, la clase `Turno` es una clase que implica cierto dinamismo al involucrar el algoritmo de alternancia.

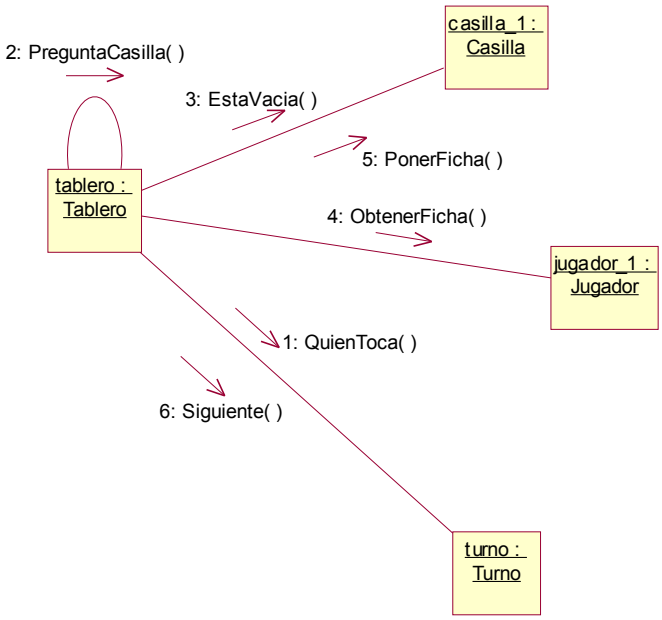


A continuación se presentan diferentes Diagramas de Interacción que ilustran cómo se relacionan las clases propuestas para conseguir el desarrollo del programa.

Como consecuencia del análisis realizado para construir estos diagramas, pueden aparecer nuevas clases que deben incorporarse al Diagrama Estático de Clases. Además, en estos diagramas aparecen funciones y propiedades que contienen las clases y que, de no estar, deben incorporarse al Diagrama Estático de Clases. Así, los diagramas se van completando unos a otros, en paralelo, hasta que finalmente el diseño es coherente.







A.5.4 Solución del ejercicio 4

```
// Source file: Ficha.java

public class Ficha {
  private Jugador propietario;

  /**
    Construye una ficha asignándole su propietario
  */
  Ficha(Jugador jugador) {
    propietario = jugador;
  }

  /**
    Devuelve al jugador propietario de la ficha
  */
  public Jugador TipoFicha() {
  return propietario;
  }
}
```

```
// Source file: Casilla.java

import java.math.*;

public class Casilla {
  private Ficha m_Ficha;
  private int fila;
  private int columna;


  /**
     Construye nuna casilla que conoce sus coordenadas en el tablero
  */
  Casilla(int fila, int columna) {
    this.fila    = fila;
    this.columna = columna;
  }

  /**
     @return True si la casilla esta vacía
  */
  public boolean estaVacia() {
    return (m_Ficha == null);
  }

  /**
     Pone una ficha en una casilla
  */
  public void ponerFicha(Ficha ficha) {
    m_Ficha = ficha;
  }

  /**
     Obtiene la ficha que hay en una casilla
  */
  public Ficha obtenerFicha() {
    return m_Ficha;
  }

  /**
     Mira si dos casillas son adyacentes
  */
  public boolean adyacente(Casilla casilla) {
    return ((Math.abs(casilla.fila - fila) < 2) &&
     (Math.abs(casilla.columna - columna) < 2));
  }
}
```

```
// Source file: Jugador.java

import Ficha;

public class Jugador {
  private Ficha m_Ficha[] = new Ficha[3];
  private int cont_fichas = 0;
  private final int NUM_FICHAS = 3;
  private String m_nombre;

  /**
     Inica un jugador con su nombre y crea sus fichas.
  */
  Jugador(String nombre) {
    m_nombre = nombre;

    for (int cont = 0; cont < NUM_FICHAS; cont++)
      m_Ficha[cont] = new Ficha(this);
  }

  /**
     Da una ficha. Desde ese momento ya no la posee.
     @return Una ficha.
  */
  public Ficha obtenerFicha() {
    if (cont_fichas < 3)
      return m_Ficha[cont_fichas++];
    else
      return null;
  }

  /** @return El nombre del jugador.
  */
  public String nombre() {
    return m_nombre;
  }
}
```

```
// Source file: Turno.java

import Jugador;

public class Turno {
  private Jugador m_Jugador[] = new Jugador[2];
  private int turno = 0;

  /**
     @return A que juagdor toca jugar
  */
  public Jugador quienToca() {
    return m_Jugador[turno % 2];
  }

  /** Cambia el turno
  */
  public void siguiente() {
    turno++;
  }

  /**
    Se construye, almacenando los jugadores e iniciando un turno
  */
  public Turno(Jugador jugador_1, Jugador jugador_2) {
      m_Jugador[0] = jugador_1;
      m_Jugador[1] = jugador_2;
    }
}
```

```
// Source file: Tablero.java

import Turno;
import Jugador;
import Casilla;
import java.io.*;

public class Tablero {
  private final int LADO = 3;
  private Casilla m_Casilla[] = new Casilla[LADO*LADO];
  private Turno m_Turno;

  /**
     Construye el tablero y prepara el turno
  */
  Tablero(Turno turno) {
    m_Turno = turno;
    for (int fila = 0;    fila    < LADO; fila++)
    for (int columna = 0; columna < LADO; columna++)
    {
      m_Casilla[fila * LADO+columna] = new Casilla(fila,columna);
    }
  }

  /**
    Devuelve la casilla correspondiente a unas coordenadas
  */
  public Casilla obtenerCasilla(int fila, int columna) {
    return m_Casilla[fila * LADO+columna];
  }

  /**
     Pregunta unas coordenadas por pantalla
  */
  private Casilla preguntaCasilla() throws Exception {
    System.out.print("Introduzca coordenda de fila: ");
    BufferedReader data = new BufferedReader(new InputStreamReader(System.in));
    int fila = Integer.parseInt(data.readLine());

    System.out.print("Introduzca coordenda de columna: ");
    int columna = Integer.parseInt(data.readLine());

    return obtenerCasilla(fila,columna);
  }

  /**
     Inicia y juega la partida controlando el turno.
  */
  public void jugarPartida() throws Exception {
    //Ponemos las primeras 5 casillas
    for (int cont_pon = 0; cont_pon < 5; cont_pon++)
    {
      ponerFicha();
      m_Turno.siguiente();
    }
```

```
    //Si ha ganado alguno fin
    if (haGanado() != null) {
        System.out.print("Ha ganado el jugador ");
        System.out.println(haGanado().nombre());
        return;
    }

    //En otro caso pedimos que se ponga la sexta ficha
    ponerFicha();
    m_Turno.siguiente();

    while (haGanado() == null) {
      moverFicha();
      m_Turno.siguiente();
    }
    System.out.print("Ha ganado el jugador ");
    System.out.println(haGanado().nombre());
  }

  /**
     Pone inicialmente las fichas en el tablero preguntando a los jugadores
   */
  public void ponerFicha() throws Exception {
    System.out.print("Jugador ");
    System.out.print(m_Turno.quienToca().nombre());
    System.out.println(" Introduzca casilla para poner ficha");

    Casilla casilla = null;
    do
      casilla = preguntaCasilla();
    while ((casilla == null) || (!casilla.estaVacia()));
    casilla.ponerFicha(m_Turno.quienToca().obtenerFicha());
  }



  /**
     Mueve una ficha desde una casilla hasta otra
  */
  public void moverFicha()  throws Exception {
    System.out.print("Jugador ");
    System.out.print(m_Turno.quienToca().nombre());
    System.out.println(" introduzca ficha a mover");

    //Que la casilla origen tenga una ficha del juagdor que toca
    Casilla casilla_origen = null;
    do
        casilla_origen = preguntaCasilla();
    while ((casilla_origen == null) ||
         (casilla_origen.estaVacia()) ||
         (m_Turno.quienToca() != casilla_origen.obtenerFicha().tipoFicha()));

  System.out.print("Jugador ");
  System.out.print(m_Turno.quienToca().nombre());
  System.out.println(" introduzca casilla destino");

  //Que la casilla destino este vacía
  Casilla casilla_destino = null;
  do
    casilla_destino = preguntaCasilla();
  while ((casilla_destino == null) ||
         (!casilla_destino.estaVacia()) ||
         (m_Turno.quienToca() != casilla_origen.obtenerFicha().tipoFicha()) ||
         (!casilla_origen.Adyacente(casilla_destino)));

  casilla_destino.PonerFicha(casilla_origen.obtenerFicha());
  casilla_origen.PonerFicha(null);
}

/**
   Comprueba si hay 3 fichas en linea del mismo jugador
   @return Devuelve el jugador ganador o en otro caso null
*/
public Jugador haGanado() {
  //Comprobación de las filas
  for (int fila = 0; fila < LADO; fila++) {
    boolean tres_en_raya = true;

    Jugador pieza = null;

    if (obtenerCasilla(fila,0).obtenerFicha() != null) {
      pieza = obtenerCasilla(fila,0).obtenerFicha().tipoFicha();
      for (int columna = 1; columna < LADO; columna++) {
        if ((obtenerCasilla(fila,columna).obtenerFicha() == null) ||
            (pieza != obtenerCasilla(fila,columna).obtenerFicha().tipoFicha()))
          tres_en_raya = false;
      }
    }
    if ((pieza != null) && (tres_en_raya))
      return pieza;
  }
```

```
  //Comprobación de las columnas
  for (int columna = 0; columna < LADO; columna++)
  {
    boolean tres_en_raya = true;

    Jugador pieza = null;
    if (obtenerCasilla(0,columna).obtenerFicha() != null) {
      pieza = obtenerCasilla(0,columna).obtenerFicha().tipoFicha();
      for (int fila = 1; fila < LADO; fila++) {
        if ((obtenerCasilla(fila,columna).obtenerFicha() == null) ||
            (pieza != obtenerCasilla(fila,columna).obtenerFicha().tipoFicha()))
          tres_en_raya = false;
      }
        if ((pieza != null) && (tres_en_raya))
          return pieza;
      }
    }

    //Comprobación de una diagonal
    boolean tres_en_raya = true;
    Jugador pieza = null;

    if (obtenerCasilla(0,0).obtenerFicha() != null) {
      pieza = obtenerCasilla(0,0).obtenerFicha().tipoFicha();

      for (int pos = 1; pos< LADO; pos++) {
        if ((obtenerCasilla(pos,pos).obtenerFicha() == null) ||
           (pieza != obtenerCasilla(pos,pos).obtenerFicha().tipoFicha()))
          tres_en_raya = false;
      }
      if ((pieza != null) && (tres_en_raya))
        return pieza;
    }

    //Comprobación de la otra diagonal
    tres_en_raya = true;
    pieza = null;

    if (obtenerCasilla(LADO-1,0).obtenerFicha() != null) {
      pieza = obtenerCasilla(LADO-1,0).obtenerFicha().tipoFicha();
      for (int pos = 1; pos < LADO;pos++) {
        if ((obtenerCasilla(pos,pos).obtenerFicha() == null) ||
            (pieza != obtenerCasilla(pos,pos).obtenerFicha().tipoFicha()))
          tres_en_raya = false;
      }
      if ((pieza != null) && (tres_en_raya))
        return pieza;
    }

    return pieza;
  }
}
```

```
// Source file: TresEnRaya.java

import Tablero;
import Turno;
import Jugador;

public class TresEnRaya {
  /** Crea objetos y lanza una partida
  */
  public static void main(String argv[]) {
    Jugador m_Jugador_1 = new Jugador("A");
    Jugador m_Jugador_2 = new Jugador("B");
    Turno m_Turno       = new Turno(m_Jugador_1, m_Jugador_2);
    Tablero m_Tablero   = new Tablero(m_Turno);

    try {
      m_Tablero.jugarPartida();
    }
    catch (Exception e) {
      System.out.println("Ha ocurrido un error inesperado");
    }
  }
}
```

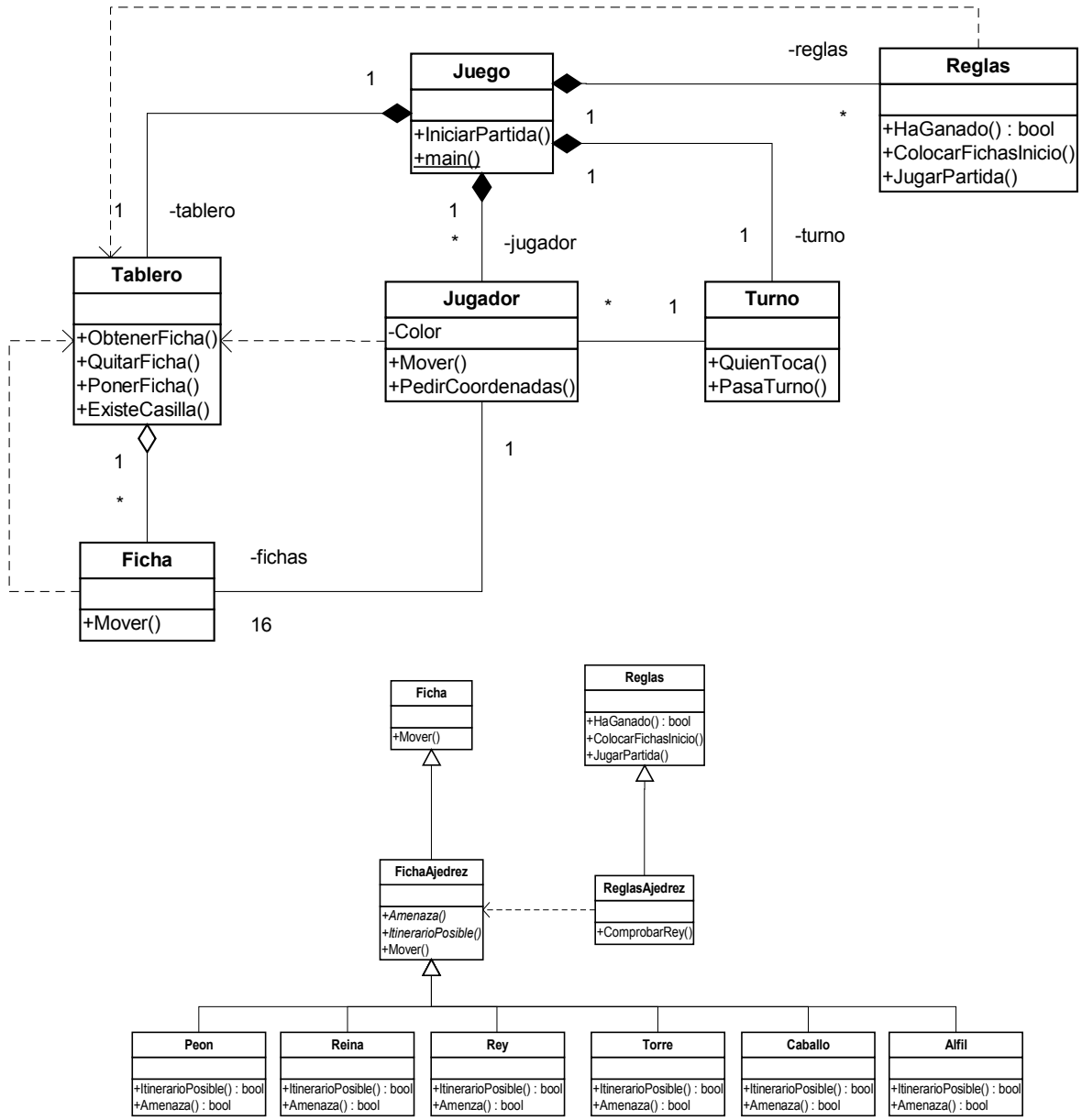
A.5.5 Solución del ejercicio 5

Se comienza realizando un diseño estático de clases, que se divide en dos grupos:

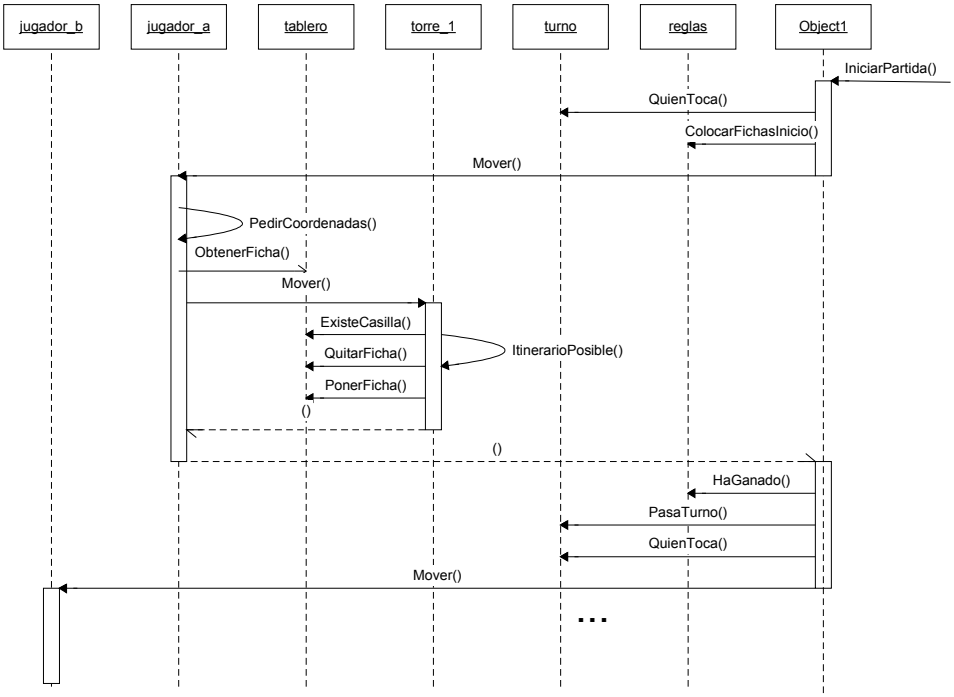
- Clases útiles para cualquier juego de tablero (`Tablero`, `Jugador`, `Turno`, `Reglas` y `Ficha`).

- Clases útiles sólo para el juego de ajedrez (`ReglasDeAjedrez`, `Peón`, `Reina`, `Rey`....).

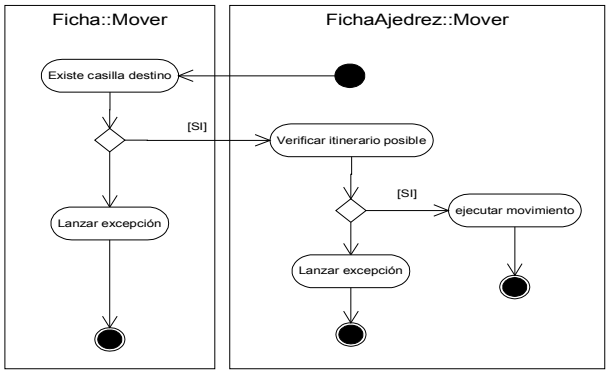
En el diagrama de la figura adjunta se aprecia como se han separado mediante herencia todos los elementos que son susceptibles de implementación especifica para el juego de ajedrez. De esta manera, luego, cuando se implemente otro juego, podrán ser sustituidos por otros con la misma interfaz pero otro comportamiento. Con esto se pretende que los objetos independientes del juego de ajedrez realicen operaciones genéricas validas para todos los juegos que luego se desee implementar.

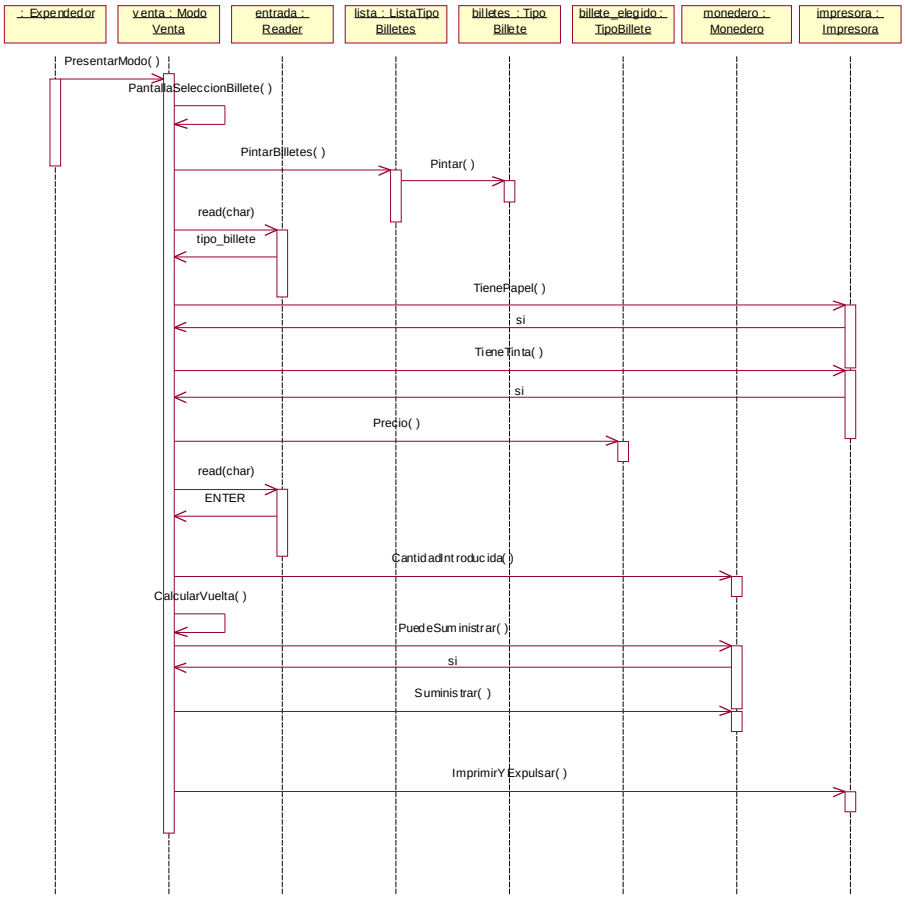


El siguiente Diagrama de Secuencia ilustra cómo se implementaría el escenario en el que un jugador pide un movimiento por pantalla, comprueba que se puede realizar y lo ejecuta, comiendo una pieza del contrario.



Para especificar en mayor detalle cómo se realiza la operación de movimiento dentro de la clase `Torre`, y como afecta a la clase `Ficha` de la que deriva, se realiza el siguiente Diagrama de Actividad.



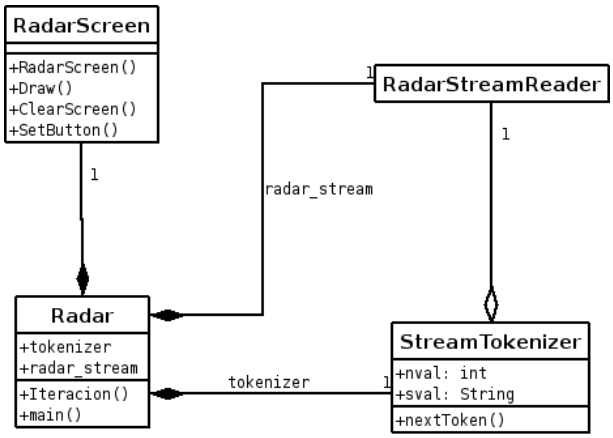


A.6 Ejercicios del capítulo 6

A.6.1 Solución del ejercicio 1

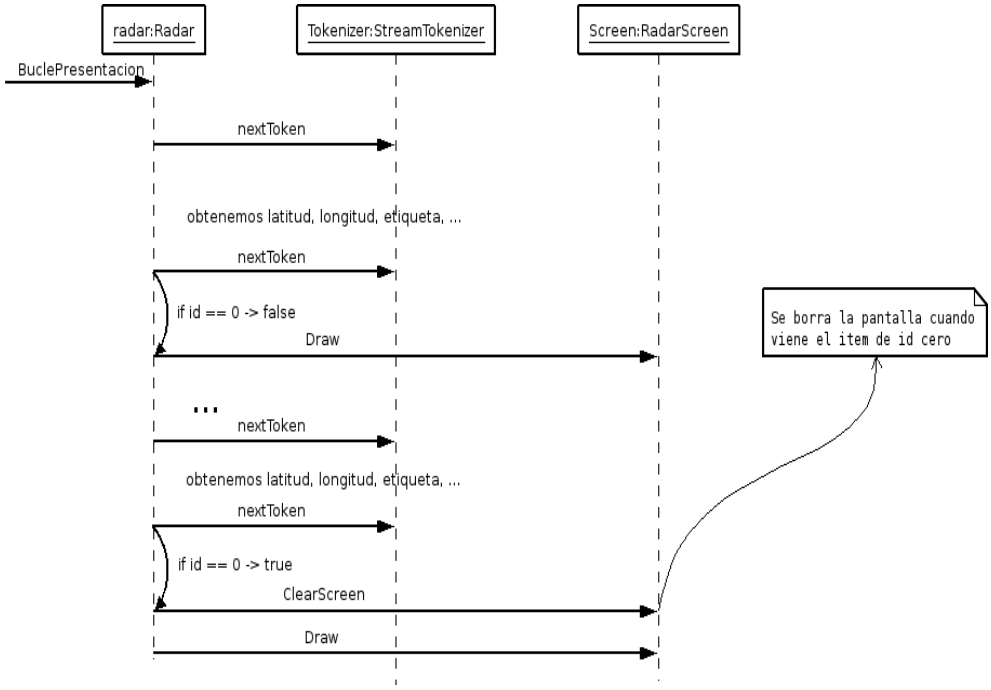
a) Una posible solución es aquella en la que sólo es necesario añadir una clase `Radar`.

La clase `Radar` se encarga de realizar el bucle de presentación. Esta clase comunica a un objeto de la clase `RadarScreen` la información que le llega del `RadarStreamReader`.



Otras posibles soluciones podrían incluir a una clase `ObjetoVolador`, cuyos objetos son capaces de leerse del *stream* y escribirse en la pantalla.

b) El Diagrama de Secuencia es:

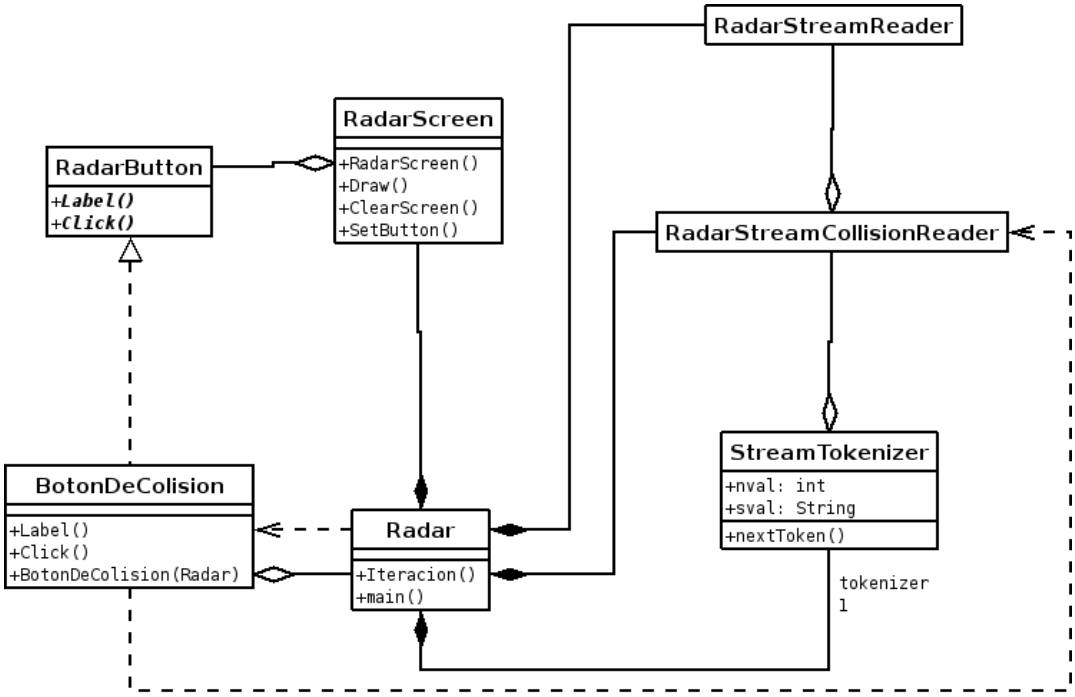


Este diagrama se transforma en los siguientes fragmentos de código:

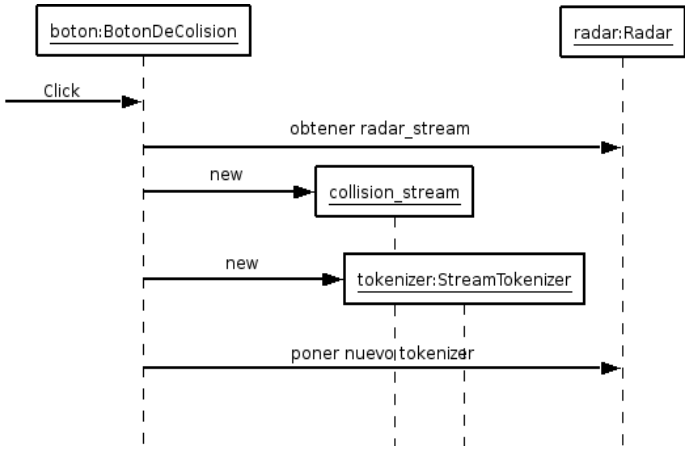
```
void buclePresentacion() {
   while(!stop) {
      nextToken();
      int identificador = nval;
      nextToken();
      int latitud = nval;
      nextToken();
      int longitud = nval;
      nextToken();
      int color = nval;
      nextToken();
      String etiqueta = sval;
      if (id == 0)
         pantalla.clearScreen();
      pantalla.Draw(latitud,longitud,color,etiqutea)
   }
}
```

Es necesario crear una clase `BotonDeColision` que implemente la interfaz `RadarButton`. Cuando se invoque `click()` la clase cambiará el `Tokenizer` para que tome los elementos desde el `RadarStreamCollisionReader` en vez de tomarlos directamente del `RadarStreamReader`. De esta forma el bucle de presentación seguirá funcionando de la misma forma.

El Diagrama Estático de Clases resultante es:



c) El Diagrama de Secuencia es:



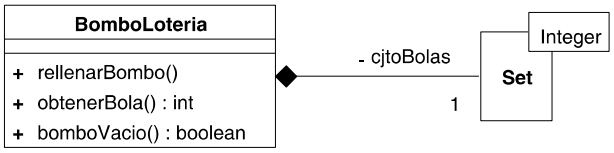
El Diagrama de Secuencia corresponde al siguiente fragmento de código:

```
void click() {
   RadarStreamCollisionReader collision_stream =
             new RadarStreamCollisionReader(radar.radar_stream);
   radar.tokenizer = new StreamTokenizer(collision_stream);
}
```

A.7 Ejercicios del capítulo 7

A.7.1 Solución del ejercicio 1

Se puede utilizar un objeto de algún derivado de `Set` para almacenar los enteros que luego se irán extrayendo hasta completar el Bingo.



Anexo B Índice alfabético

A

Abierto-cerrado, 100
Abstract, 30, 31
Abstract Factory, 93
Abstraer, 2, 4
Acoplamiento, 5
Adaptador, 95
Adapter, 95
Ámbitos, 21
Array, 33
ArrayList, 87
Asociación, 8
Autoboxing, 50, 61
Autounboxing, 50, 61

B

Barrera de la abstracción, 6
Bloqueado, 58
Bloqueos mutuos, 57
Bloques de inicialización, 23
Boolean, 19
Break, 23
Builder, 94
Byte, 19
Bytecode, 16

C

Caja de arena, 16
Calles, 74
Camello, 18
Campos, 25
Cardinalidad, 9
Case, 23
Casting, 11, 19, 41
Catch, 51
Char, 19
Checked exceptions, 52
Clase, 4
Clase base, 9
Clase Class, 49
Clase de objetos, 4
Clase derivada, 9
Clase hija, 9
Clase padre, 9
Clases anónimas, 32
Clases de envoltura, 61
Clases internas, 23, 31, 38
Clases parametrizadas, 44
Class, 24, 49
CLASSPATH, 34
Cliente/servidor, 4
Cloneable, 55
Código fuente, 16
Código máquina, 16
Coerción, 11
Coherencia, 10
Cohesión, 5
Collection, 87
Collections, 90
Comando, 98
Command, 98
Comparable, 57
Complejidad, 1
Comportamiento, 3
Composición, 97
Composite, 97
Concurrencia, 13, 57
Condiciones de carrera, 57
Constructor, 28, 38
Constructor por defecto, 28
Continue, 23
Contrato, 4
Conversión, 11

D

Deadlocks, 57
Decorador, 95
Decorator, 95
Default, 23
Dependencia, 8

Dependencias, 10
Descomponer, 1
Descomposición algorítmica, 1
Descomposición orientada a objetos, 2
Deserialización, 84
Diagrama de Actividad, 73
Diagramas de Colaboración, 64
Diagramas de Instancias, 10
Diagramas de Interacción, 64
Diagramas de Paquetes, 10
Diagramas de Secuencia, 64
Diagramas Estáticos de Clases, 5
Do while, 23
Double, 19

E

Economía, 64
Eficiencia, 64
Ejecutable, 58
Else, 22
Encapsular, 5
Enlace tardío, 12
Enlace temprano, 12
EnumSet, 88
Envoltorio, 95
Envolturas, 61
Error, 52
Estado, 3
Estrategia, 97
Estructurado, 3
Excepción, 4, 51
Excepciones controladas, 52
Excepciones no controladas, 52
Exception, 27, 52
Extends, 37

F

Fábrica Abstracta, 93
Factory Method, 94
Final, 28, 30, 31
Finalizadores, 29
Finally, 51
Float, 19
For, 22, 90
Fork, 13
Friendly, 30

G

Garbage collector, 29
Genericidad, 44
Genérico, 44

H

HashMap, 89
HashSet, 87
Herencia, 7, 37
Herencia múltiple, 7, 38
Herencia simple, 7, 37
Hilos, 13

I

Identidad, 3
If, 22
Implements, 43
Import, 34
Inanición, 57
Inmutable, 18, 51, 90
InputStream, 79
Instanceof, 41
Int, 19
Interfaces, 4, 42
Iterador, 97
Iteradores, 90
Iterator, 97

J

Java Native Interface, 31
Javadoc, 17
Jerarquizar, 2, 6
JIT, 16
JNI, 31
Just In Time, 16

L

Lenguaje Unificado de Modelado, 4
Líneas de ejecución, 13
LinkedList, 87
Liskov, 101
List, 87
Long, 19

M

Map, 87, 89
Máquina Virtual, 16
Métodos, 4
Miembros, 4
Modelo de Objetos, 3
Modificadores en la definición de clases generales, 31
Modularizar, 10
Módulos, 10
Monitor, 58
Muerto, 58

N

Native, 31
New, 28
Nombres completamente calificados, 34
Notify, 59
NotifyAll, 59
Nuevo, 58
Null, 25

O

Object, 49
Objeto, 3
Observador, 100
Observer, 100
Ocultación, 5
Operadores, 20
Orientación a aspectos, 3
OutputStream, 79

P

Paquete, 10
Paquete por defecto, 35
Paradigma imperativo, 3
Patrón Comando, 98
Patrón Composición, 97
Patrón Constructor, 94
Patrón Decorador, 95
Patrón Envoltorio, 95
Patrón Estrategia, 97
Patrón Fábrica, 94
Patrón Iterador, 97
Patrón Método de Construcción, 94
Patrón Observador, 100
Patrones de Diseño, 93
Persistencia, 13
Polimorfismo, 12
Polimorfismo dinámico, 12, 39
Polimorfismo estático, 12, 28
Poscondiciones, 4
Precondiciones, 4
Primitividad, 5
Principio abierto-cerrado, 100
Principio de inversión de dependencia, 101
Principio de Liskov, 101
Principio de responsabilidad única, 101
Principio de segregación de interfaces, 101
Private, 6, 30, 31
Problema de la ambigüedad, 7
Problema del diamante, 7
Propiedades, 4
Protected, 6, 30, 31
Protocolo, 4
Public, 6, 30, 31

R

Recolector de basura, 29
Redefinición, 38
Reflexión, 49
Región de exclusión mutua, 58
Relaciones de asociación, 8
Relaciones de dependencia, 8
Relaciones de uso, 8
Return, 27
Run, 57

S

Segregación de interfaces, 101
Sentencias, 21
Serializable, 84
Serialización, 79, 84
Set, 87
Short, 19
Sleep, 58
Sobrecarga, 28
SortedMap, 89
SortedSet, 87
Stack, 87
Start, 57

Starvation, 57
Static, 30, 31
Strategy, 97
Stream, 79
Strictfp, 31
Suficiencia y completitud, 5
Super, 38
Superclase, 9
Switch, 23
Synchronized, 31, 58

T

Tamaño de los módulos, 10
Temporalidad, 63
This, 26
Thread, 57
Threads, 13
Throw, 27, 51
Throwable, 51
Throws, 27
Tipado, 11
Tipado débil, 11
Tipado dinámico, 12
Tipado estático, 12
Tipado explicito, 12
Tipado fuerte, 11
Tipado implícito, 12
Tipado latente, 12
Tipo, 11
Tipos enumerados, 59
Tipos parametrizados, 47
Tipos primitivos, 18
Transient, 31, 85
TreeMap, 89
TreeSet, 87
Try, 51

U

UML, 4
Unchecked exceptions, 52
Unified Modeling Language, 4
Uso, 8

V

Variables, 18
Vector, 87
Versatilidad, 63
Visibilidad, 63
Volatile, 31

W

Wait, 59
While, 22
Wrapper, 95
Writer, 81