USERS

¡INCLUYE EJEMPLOS REALES!

JAVA

DOMINE EL LENGUAJE LÍDER EN APLICACIONES CLIENTE-SERVIDOR



- CONCEPTOS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS
- TDD: TEST DRIVEN DEVELOPMENT
- TÉCNICAS DE DISEÑO Y MODELADO
- PATRONES DE DISEÑO
- REFLEXIÓN Y METAPROGRAMACIÓN

por Ignacio Vivona

MANUALES USERS MANUALES USERS MANUALES USERS MANUALES USERS MAN

DESARROLLO PROFESIONAL MULTIPLATAFORMA

CONÉCTESE
CON LOS MEJORES LIBROS DE COMPUTACIÓN

LLEGAMOS A TODO EL MUNDO
VÍA OCA* Y DHL**

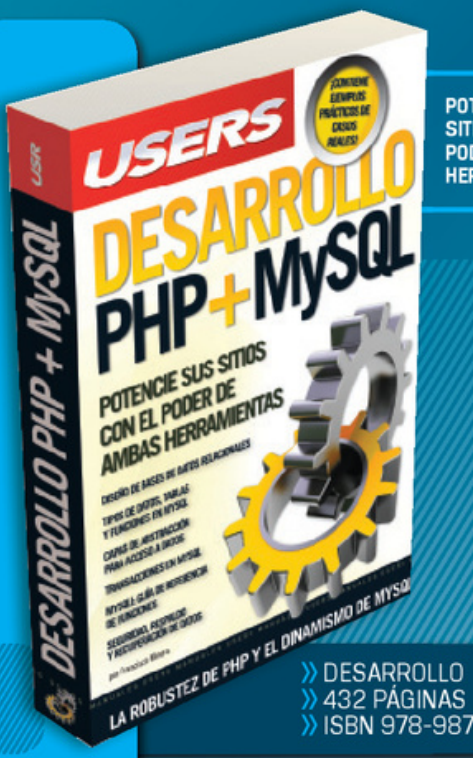
usershop.redusers.com
usershop@redusers.com

*SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // **VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA



CLAVES Y TÉCNICAS PARA OPTIMIZAR SITIOS DE FORMA PROFESIONAL

» DESARROLLO / INTERNET
» 400 PÁGINAS
» ISBN 978-987-1773-09-1



POTENCIE SUS SITIOS CON EL PODER DE AMBAS HERRAMIENTAS

» DESARROLLO
» 432 PÁGINAS
» ISBN 978-987-1773-16-9



PROGRAMACIÓN DE SITIOS WEB PROFESIONALES

» DESARROLLO / INTERNET
» 400 PÁGINAS
» ISBN 978-987-1773-07-7



DISEÑO DE SITIOS WEB COMPLEJOS CON LAS HERRAMIENTAS DE ADOBE

» DISEÑO / DISEÑO WEB
» 320 PÁGINAS
» ISBN 978-987-663-022-1

JAVA

DOMINE EL LENGUAJE LÍDER EN APLICACIONES CLIENTE-SERVIDOR

por Ignacio Vivona

RedUSERS



TÍTULO: Java

AUTOR: Ignacio Vivona

COLECCIÓN: Manuales USERS

FORMATO: 17 x 24 cm

PÁGINAS: 320

Copyright © MMXI. Es una publicación de Fox Andina en coedición con DALAGA S.A. Hecho el depósito que marca la ley 11723. Todos los derechos reservados. Esta publicación no puede ser reproducida ni en todo ni en parte, por ningún medio actual o futuro sin el permiso previo y por escrito de Fox Andina S.A. Su infracción está penada por las leyes 11723 y 25446. La editorial no asume responsabilidad alguna por cualquier consecuencia derivada de la fabricación, funcionamiento y/o utilización de los servicios y productos que se describen y/o analizan. Todas las marcas mencionadas en este libro son propiedad exclusiva de sus respectivos dueños. Impreso en Argentina. Libro de edición argentina. Primera impresión realizada en Sevagraf, Costa Rica 5226, Grand Bourg, Malvinas Argentinas, Pcia. de Buenos Aires en IX, MMXI.

ISBN 978-987-1773-97-8

Vivona, Ignacio

Java. - 1a ed. - Buenos Aires : Fox Andina; Dalaga, 2011.

320 p. ; 24x17 cm. - (Manual users; 218)

ISBN 978-987-1773-97-8

1. Informática. I. Título

CDD 005.3



ANTES DE COMPRAR

EN NUESTRO SITIO PUEDE OBTENER, DE FORMA GRATUITA, UN CAPÍTULO DE CADA UNO DE LOS LIBROS EN VERSIÓN PDF Y PREVIEW DIGITAL. ADEMÁS, PODRÁ ACCEDER AL SUMARIO COMPLETO, LIBRO DE UN VISTAZO, IMÁGENES AMPLIADAS DE TAPA Y CONTRATAPA Y MATERIAL ADICIONAL.

RedUSERS
COMUNIDAD DE TECNOLOGIA

redusers.com

Nuestros libros incluyen guías visuales, explicaciones paso a paso, recuadros complementarios, ejercicios, glosarios, atajos de teclado y todos los elementos necesarios para asegurar un aprendizaje exitoso y estar conectado con el mundo de la tecnología.



LLEGAMOS A TODO EL MUNDO VÍA OCA* Y DHL**

*SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // **VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

usershop.redusers.com // usershop@redusers.com

Ignacio Vivona



Desarrollador de software desde hace casi diez años y entusiasta de la tecnología desde hace mucho más. Ha trabajado en varios sistemas informáticos de gran envergadura, desde programas de consulta de autopartes para sistemas **GIS** de seguimiento de vehículos, hasta desarrollos para motores de reglas y sitios web de distinta índole. Ha utilizado un amplio espectro de tecnologías y plataformas para aplicar las distintas opciones y herramientas que más se adecuan a cada situación. Trabajó en plataformas **Windows** y **Linux**, haciendo uso de tecnologías como **C++** y **Visual Basic** para la creación de aplicaciones de escritorio, servicios y para componentes transaccionales. Por otra parte, también ha desarrollado diversas aplicaciones para la Web utilizando **Java**, **.NET**, **PHP** y **ASP**. Participó en la creación de varios **frameworks**, en los cuales se basaron distintas aplicaciones.

Realizó su tesis para la carrera de Licenciatura en Ciencias de la Computación en la Universidad de Buenos Aires.

Agradecimientos

A Yayi, por su confianza y su apoyo incondicional.

A Nicolás Kestelboim, por la oportunidad de escribir este libro.

A Mati, Marce y Emi por bancarme en esta iniciativa.

A mis padres y mi hermano.

Y a mis amigos.

Prólogo



Leí por primera vez algo sobre Java allá por el año 1998, en una revista de informática española. En esa ocasión me intrigó el slogan "compilarlo una vez, correrlo en cualquier lado" y desde ese momento comencé a investigar sobre el tema. Conseguí un libro sobre Java, uno de los primeros en nuestra lengua, desarrollando ejercicios y leyendo en los sitios web fui creciendo en el lenguaje.

Luego me fui metiendo cada vez más en la teoría del paradigma de la programación orientada a objetos, fui entendiendo sus bases, su historia y las ideas que propone. Esto me fue llevando a investigar otros temas, otros paradigmas y otros lenguajes.

Es esa sensación de necesitar saber cómo funciona algo, qué partes lo componen y de qué forma interactúan para cumplir con una tarea más compleja que la que realizan individualmente. Es esta curiosidad, sumada a las ganas de crear, de concretizar ideas, los elementos necesarios para ser un programador.

En este libro me propongo llevar al lector en un viaje por el maravilloso mundo de la programación con objetos. Utilizaremos como medio de transporte el lenguaje Java, uno de los exponentes más famosos de este paradigma de programación. Iremos aprendiendo las ideas y las metodologías sobre la programación con objetos, aplicando todo lo que veamos en los ejercicios y poniendo en práctica los test unitarios.

Invito ahora al lector a comenzar este recorrido juntos, e ir aprendiendo cómo materializar nuestras ideas, utilizando Java y la programación orientada a objetos. Sin más, arrancamos...

Ignacio Vivona

El libro de un vistazo

Este libro realiza un recorrido por la teoría tras las ideas soportadas por el paradigma de la programación orientada a objetos y su aplicación con Java. Los primeros capítulos presentan los conceptos del paradigma, luego se explaya en el lenguaje y en el modelado correcto con él.

*01 PROGRAMACIÓN ORIENTADA A OBJETOS

En este capítulo veremos la historia del paradigma de la orientación a objetos. Analizaremos conceptos como polimorfismo y herencia. También veremos los tipos y subtipos.

*02 INICIACIÓN A JAVA

Este capítulo se dedica a la presentación del lenguaje Java. Empezaremos viendo su historia y los motivos detrás de su creación, para luego preparar nuestros códigos.

*03 SINTAXIS

Aprenderemos cómo están compuestos los programas en Java. Veremos las distintas estructuras del lenguaje, su significado así como también las reglas que lo definen.

*04 CLASES

En este capítulo veremos las clases. Aprenderemos cómo se definen, cómo se especifica el estado de las instancias y cómo se declara el comportamiento de estas.

*05 MAS CLASES

Continuamos con el tema del capítulo anterior y avanzamos sobre temas como clases abstractas y clases anidadas. Luego presentamos un gran ejercicio: El juego de la vida.

*06 INTERFACES

En este capítulo analizaremos las interfaces. Veremos cuál es el motivo de su existencia, cómo se definen y cuándo se deben utilizar. Finalmente las compararemos con las clases abstractas.

*07 ENUMERACIONES

Aquí conoceremos las enumeraciones como la forma que tiene el lenguaje Java de definir una serie de elementos conocidos y que comparten un comportamiento y están agrupados.

*08 EXCEPCIONES

Aprenderemos en este capítulo cómo se encuentran clasificadas las excepciones, cómo funcionan, en qué casos debemos utilizarlas y de qué forma las implementamos.

*09 GENÉRICOS

En este capítulo seremos presentados con la forma que define el lenguaje que permite parametrizar los tipos utilizados en nuestro código. Veremos por qué son necesarios, cómo se definen y de qué forma debemos utilizarlos.

*10 LIBRERÍA BASE

Aquí nos encargaremos de ver la librería base que proporciona Java. Conoceremos y analizaremos las clases e interfaces principales que todo programador debe conocer.

*11 ANOTACIONES

Trataremos en este capítulo una forma de definir información extra en nuestros programas, que podrá ser consultada luego en distintos momentos de la vida de nuestro código.

*12 TÉCNICAS Y DISEÑO

Capítulo donde aprenderemos algunas técnicas de diseño que no ayudan en la tarea de construcción de software robusto, claro y mantenible. Tendremos, también, una primera aproximación a los famosos patrones de diseño.

*Ap1 REFLEXIÓN

En este apartado estudiaremos el aspecto del lenguaje Java que permite analizar y modificar las clases y también los objetos desde el mismo programa mientras es ejecutado.

*Ap2 SIGUIENTES PASOS

Finalmente, en este anexo, se nos presentarán distintos caminos que podremos seguir para ir enriqueciendo y ampliando nuestro conocimiento sobre Java y el desarrollo de software.

INFORMACIÓN COMPLEMENTARIA

A lo largo de este manual podremos encontrar una serie de recuadros que nos brindarán información complementaria: curiosidades, trucos, ideas y consejos sobre los temas tratados. Para poder reconocerlos fácilmente, cada recuadro está identificado con diferentes iconos:


CURIOSIDADES E IDEAS


ATENCIÓN


DATOS ÚTILES Y NOVEDADES


SITIOS WEB

RedUSERS

MEJORA TU PC



Desarrollos temáticos en profundidad

Libros.

Coleccionables.

Cursos intensivos con multimedia





Capacitación dinámica

Revistas.

Sitios Web.

Noticias al día, downloads, comunidad





Información actualizada al instante

Newsletters.

La red de productos sobre tecnología más importante del mundo de habla hispana.

redusers.com

Contenido

Sobre el autor 4
Prólogo 5
El libro en un vistazo 6
Información complementaria 7
Inroducción 12

*01 Programación Orientada a Objetos

Historia 14
Conceptos 15
Terminología 17
Polimorfismo 18
Tipos y Subtipos 19
Herencia 21
Herencia simple versus herencia múltiple 24
¿Por qué objetos? 25
Resumen 27
Actividades 28

*02 Iniciación a Java

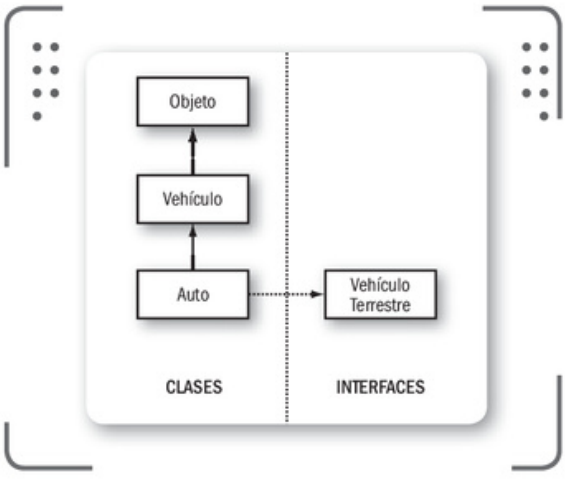
Historia 30
Preparación 32
Eclipse IDE 33
Primeros códigos 38
Resumen 43
Actividades 44

*03 Sintaxis

Palabras claves 46
Ciclos 52
Declaraciones, expresiones, sentencias y bloques 58
Variables 58
Sentencias 60
Otras estructuras 60
if/else/else if 60
Switch 63
Try/catch/finally 65
Tipos primitivos y literales 67
Operadores 70
Paquetes 72
Resumen 73
Actividades 74

*04 Clases

Definición 76
Atributos 79
Métodos 82
La herencia y los métodos 84
Constructores 85
This y Super 87
Lo estático versus lo no estático 90
Resumen 91
Actividades 92



*05
Más clases

Clases abstractas **94**
Clases anidadas 97
Clases anidadas estáticas 97
Clases internas 98
Clases locales y clases anónimas **101**
Ejercicio: El juego de la vida **103**
Resumen **121**
Actividades **122**

*06
Interfaces

Definición **124**
Uso **126**
Clases abstractas versus interfaces **129**
Resumen **131**
Actividades **132**

*07
Enumeraciones

Definición **134**
Uso **136**
Resumen **147**
Actividades **148**

*08
Excepciones

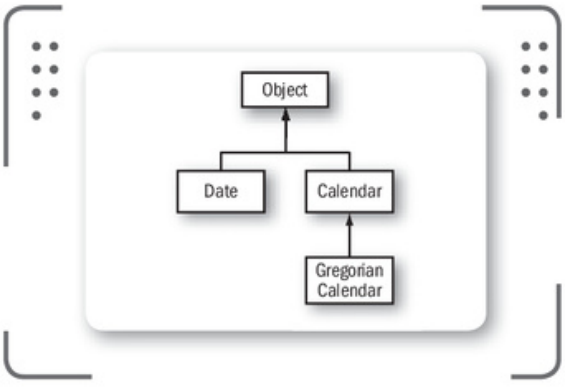
Definición **150**
Uso **154**
Excepciones chequeadas **157**
Excepciones no chequeadas **158**
Errores **159**
Resumen **159**
Actividades **160**

*09
Genéricos

Definición **162**
Subtipado **166**
Comodín **166**
Tipos restingidos **167**
Genéricos en el alcance estático **168**
Resumen **169**
Actividades **170**

*10
Librería Base

Librería y objetos básicos **172**
java.lang.Object 172
java.lang.Boolean 174
Colecciones **177**
java.util.Collection 178
java.util.Set 180
java.util.Map 181
Ejercicio: colecciones diferentes **182**
Clases útiles **186**
Ejercicio: alternativa a java.util.Date 188
I/O **192**
java.io.InputStream y su familia 192
java.io.Reader y java.io.Writer 195
Resumen **197**
Actividades **198**

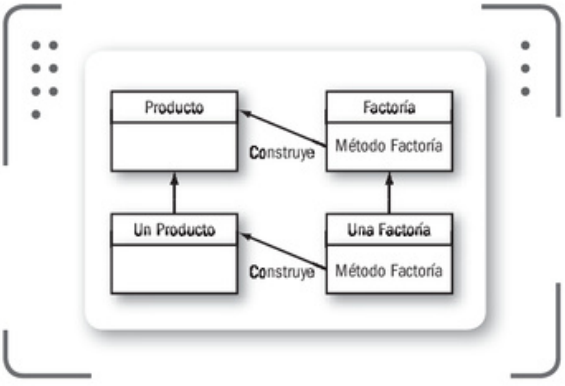


*11 Anotaciones

¿Qué son las anotaciones?....200
Definición....204
Heredando anotaciones....208
Uso de las anotaciones....209
Accediendo a las anotaciones en tiempo de ejecución....209
Jerarquizando anotaciones....212
Distintos usos de las anotaciones....217
Inyección de dependencias....219
Serialización....220
Mapeos a base de datos....222
Aplicaciones web....223
Resumen....223
Actividades....224

*12 Técnicas y Diseño

Inmutabilidad, Java Beans y la creación de objetos....226
Inyección de dependencias e inversión de control....233
Sobre la creación de objetos....237
Método factoría....237
Factoría abstracta....239
Singleton....241



Evitar utilizar null....243
Representar conceptos con objetos....245
Resumen....247
Actividades....248

*Ap1 Reflexión

¿Qué es la reflexión?....250
Las clases....251
Métodos....255
Constructores....261
Atributos....262
Modificadores....264
Interfaces....265
Clases anidadas....265
Arrays....268
Los ClassLoaders....277
Resumen....277
Actividades....278

*Ap2 Siguientes pasos

Temas para seguir estudiando....280
Desarrollo de aplicaciones para la consola de comandos....296
Desarrollo de aplicaciones de escritorio....298
Desarrollando aplicaciones para dispositivos móviles....301
Desarrollando aplicaciones web....303
Otros lenguajes para la JVM....307
Resumen....308

* Servicios al lector

Índice temático....310

Introducción

¿Cómo funciona esto? ¿Es posible hacer algo así? Estas son preguntas que todo aquel que se interesa por la tecnología y la computación se hace. Y estas inquietudes son las que nos llevan a embarcarnos en el camino de la programación. Hoy en día las computadoras están en todos lados y en todas partes. Cualquiera tiene los medios y la posibilidad de responder las dudas anteriores y de crear programas y herramientas que permitan mejorar el mundo. Cualquiera puede ser el autor de la próxima gran idea millonaria.

La programación orientada a objetos ha dominado el mercado por sobre otros paradigmas durante los últimos veinte años, y parece que va seguir dominándolo por mucho tiempo más. En particular Java es el máximo referente de los programadores y existe una comunidad que constantemente genera productos, librerías y frameworks. Aun así, la demanda de nuevos programadores sigue creciendo. La orientación a objetos encabeza la difícil tarea de transmitir ideas y de plasmarlas en la PC, de tal forma que un sistema pueda hacer lo que queramos de una manera sencilla y tangible.

En este libro, el lector tendrá la oportunidad de sumergirse en el paradigma; tomar sus ideas, sus reglas y sus metodologías. Luego, todos estos conocimientos se aplicarán a un lenguaje particular, en este caso, Java. Irá conociendo y aprendiendo el lenguaje paso a paso, desde su sintaxis y semántica, para escribir código correcto y entender cómo plasmar y modelar adecuadamente las situaciones de nuestro interés. Para esto se usarán las herramientas provistas por el lenguaje, como las clases y la herencia, y también las interfaces y conceptos que provee Java. El lector está invitado a continuar, con la mente abierta a nuevas ideas, sin preconceptos ni prejuicios. Suerte.

Ignacio Vivona

*01

Programación orientada a objetos

Conocer los conceptos básicos que corresponden a este paradigma de programación es fundamental para poder entender y aplicar correctamente los conocimientos que iremos adquiriendo a lo largo del libro. En este capítulo nos encargaremos de revisar y entender los alcances de la programación orientada a objetos y su relación con Java.

- **Historia** 14
- **Conceptos** 15
  - Terminología 17
  - Polimorfismo 18
- **Tipos y Subtipos** 19
- **Herencia** 21
  - Herencia simple versus herencia múltiple 24
- **¿Por qué objetos?** 25
- **Resumen** 27
- **Actividades** 28

Historia

Las raíces de la programación orientada a objetos se remontan a los años 60, con lenguajes como **Algol 60** y **Simula**. Pero no es sino hasta los 70, cuando **Alan Kay** y su equipo empiezan a trabajar en la creación del lenguaje **Smalltalk**, el momento en el cual se acuña el término **orientado a objetos**. El lenguaje **Smalltalk** fue el pilar de este tipo de programación y, actualmente, es uno de los pocos lenguajes puros, donde todo es un objeto. Se le dice puro a un lenguaje cuando todos sus elementos son objetos. Muchos lenguajes de este tipo no son puros (como Java) porque tienen elementos, como los números, que no lo son. Esto crea una discontinuidad en el razonamiento, dado que hay que pensar en objetos y también tener en cuenta los elementos que no son objetos. Un lenguaje puro tiene un concepto uniforme, todos son objetos, y así facilita pensar los problemas y sus soluciones.

Alan Kay y su equipo trabajaban en Xerox PARC (un centro de investigación y desarrollo) y fueron pioneros en varias tecnologías que hoy en día son comunes. El sistema en el que trabajaban se denominaba **Smalltalk**, y se caracterizaba por ser un sistema operativo más que un lenguaje de programación. Fue el primero en tener ventanas, barras de desplazamiento, botones y hacer uso del mouse (aunque no fue inventado por ellos). Es un hecho que Steve Jobs, en una de sus visitas al Xerox PARC, vio **Smalltalk** y así surgió la Apple Lisa, la primera computadora personal con interfaz gráfica.

Smalltalk innovó también en ser un lenguaje dinámico y re□ectivo donde un usuario podía modificar el comportamiento de cualquier parte del sistema en tiempo de ejecución (no había un tiempo de compilación como en otros lenguajes), de esta manera el sistema estaba constantemente vivo. Las capacidades re□ectivas permitían razonar y manipularlo dinámicamente mientras estaba corriendo.



OTROS APORTES DE SMALLTALK

De la comunidad Smalltalk también surgieron adelantos como los **test unitarios** (que veremos más adelante) y los **refactorings** (cambios que mejoran el código sin cambiar el comportamiento de éste), dos temas que han adquirido gran importancia en los últimos años. ¡Y surgieron hace veinte!

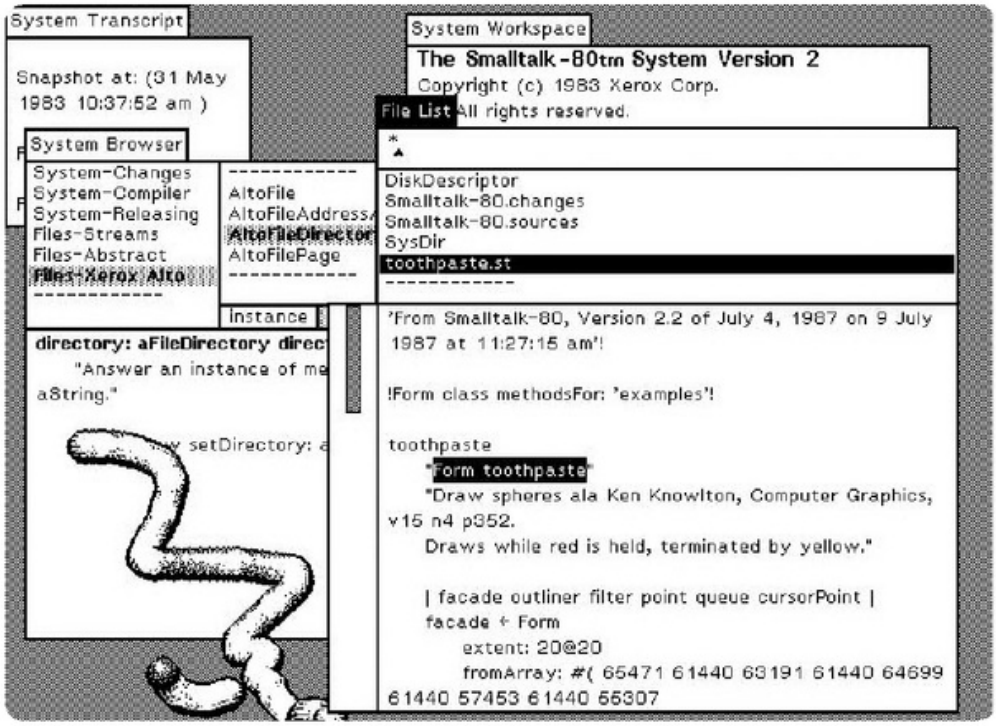


Figura 1. Así se veía la versión 80 de Smalltalk. Contaba con la posibilidad de manipular ventanas, iconos y otros elementos. Debemos saber que fue uno de los primeros sistemas en utilizar el mouse.

Conceptos

La programación orientada a objetos se basa en dos conceptos básicos:

- Hay objetos
- Solamente se comunican enviándose mensajes

Un objeto es una entidad que agrupa datos y funcionalidad. En este esquema los datos son otros objetos, que son referenciados a través de un nombre, una etiqueta. Por ejemplo, un objeto **Persona** tiene como datos el **nombre** y la **edad**, estas son etiquetas que apuntan a los objetos correspondientes. Entendemos funcionalidad por lo que hace un objeto cuando recibe un mensaje, determinando su comportamiento. Siguiendo con el ejemplo de la persona, si se le envía el mensaje **correr**, esta empieza a correr. Esta idea difiere de otros lenguajes como **C**,

donde los datos y las funciones que operan sobre esos datos están separados. Al estar contenidos en un objeto tanto los datos como la funcionalidad, se define así una frontera entre el interior y el exterior de este.

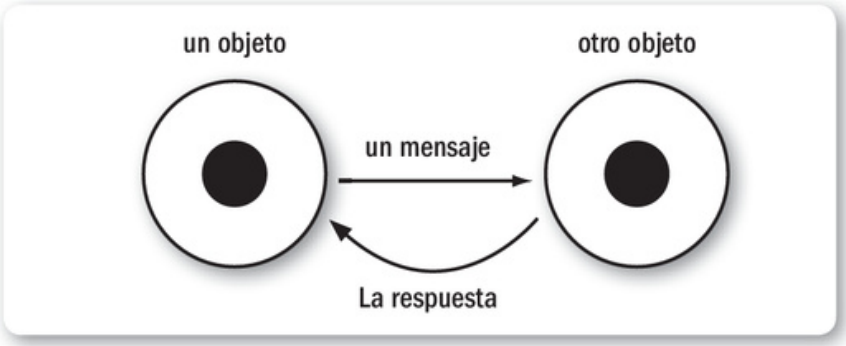


Figura 2. Vista conceptual de objetos comunicándose. Los objetos interactúan enviándose mensajes, los cuales pueden llevar más información (otros objetos) con ellos.

Esta separación aísla tanto al interior de un objeto de otros objetos, como así también de sus asuntos internos . Este aislamiento favorece la reutilización (de conocimiento, secundariamente de código), en la adaptación a los cambios y la simplicidad. Todos estos beneficios surgen por la existencia de esta frontera y porque la única forma posible de cruzarla sea el envío de mensajes. Se dice que un objeto **encapsula** datos y funcionalidad y que, además, **oculta** su información, dado que solamente expone lo que quiere hacer público a otros objetos.

El paradigma no explica nada acerca de su implementación, de cómo son los objetos ni de qué forma se envían los mensajes. Tampoco hace mención a clases ni a métodos. Al ser tan simple y no imponer restricciones, existen diversos tipos de lenguajes orientados a objetos.

Hay lenguajes que utilizan clases para agrupar y crear objetos, como en el caso de **Java**, otros utilizan **prototipos**, donde los objetos se copian de otros objetos base y, a su vez, toman cosas prestadas de ellos, como en el caso de **Javascript**. Algunos lenguajes como **Java**, **Smalltalk** y **C++,** ocultan el hecho de que se están enviando mensajes, por temas de velocidad y simplicidad. Otros, en cambio, lo hacen explícito, por temas de paralelismo, y no se espera que la respuesta sea inmediata,

como **Erlang**. A partir de estos conceptos básicos, se desprenden algunos otros que, generalmente, están asociados a la programación orientada a objetos: polimorfismo, tipos, subtipos y herencia.

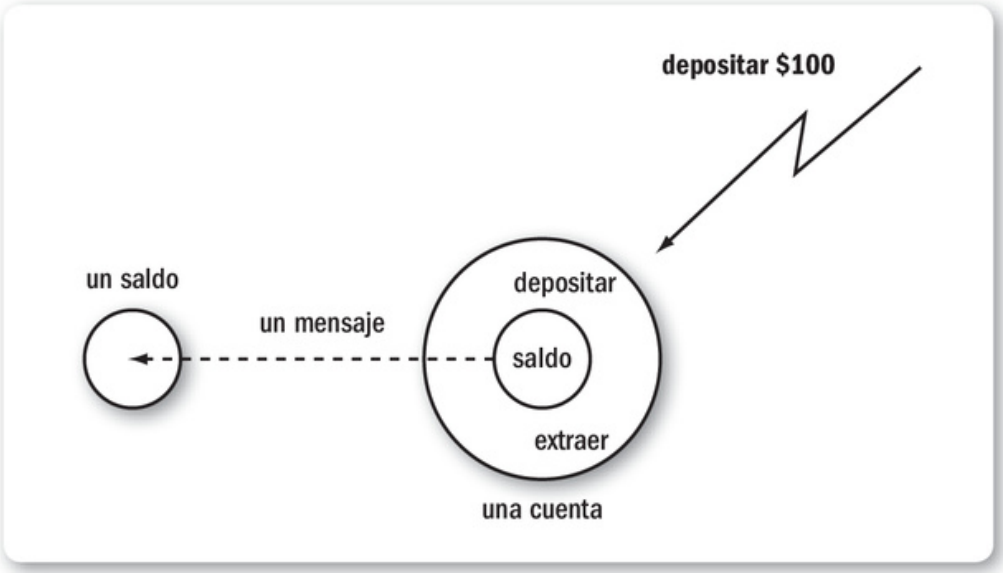


Figura 3. Vemos cómo un objeto es usado por muchos otros y cómo también puede utilizar distintos objetos, tomando como ejemplo los conceptos de encapsulamiento y ocultamiento de la información.

Terminología

Dejemos claro algunos términos antes de seguir. Llamaremos a la parte externa de un objeto, es decir, su frontera con el exterior, **interfaz** o **protocolo**. A los objetos, instancias, y cuando estemos en el contexto de clases, diremos que un objeto es **instancia de** una determinada clase. Una **clase** es un artefacto que proveen los lenguajes para construir **tipos**, que especifican el comportamiento de las instancias y las construyen. La forma en que un objeto responde a un mensaje es a través de su **implementación**, a la que llamaremos **método**. Los mensajes pueden llevar consigo objetos, a estos los denominaremos **colaboradores externos** (al objeto receptor), y también **parámetros** o **argumentos** del método. Debemos tener en cuenta que a los datos de un objeto, los que conoce por un nombre, los mencionaremos como **colaboradores internos** o **variables de instancia**.

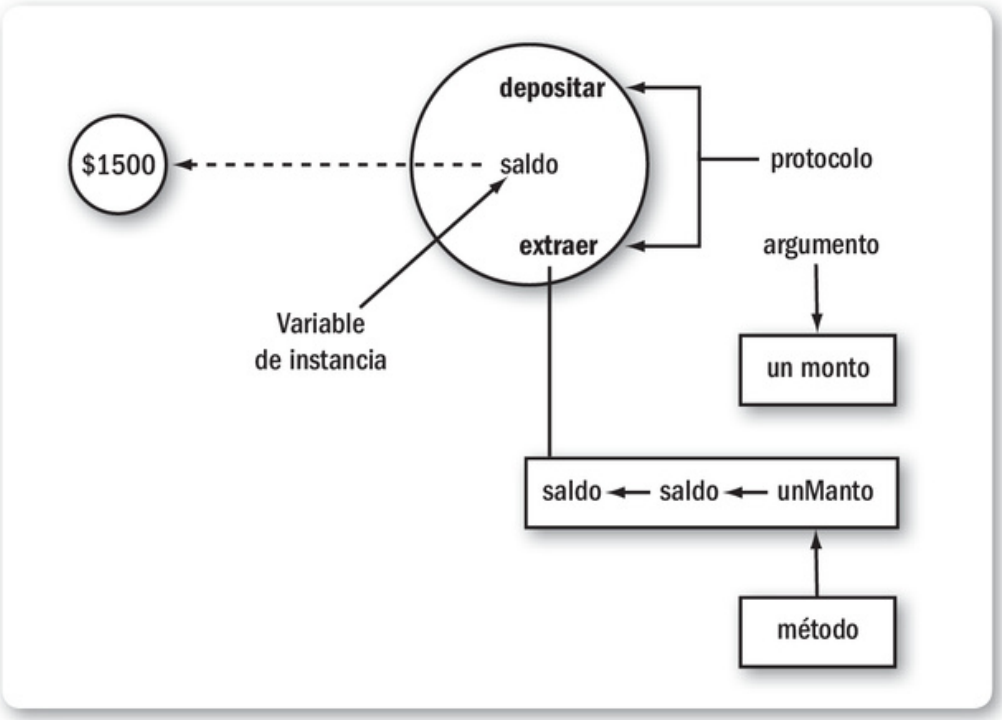


Figura 4. Un objeto reconoce a otros por medio de nombres. El protocolo establece qué mensajes entiende un objeto, y estos mensajes son los únicos elementos que permiten cruzar la frontera del objeto.

Polimorfismo

El polimorfismo es la característica por la cual distintos objetos pueden ser intercambiados entre sí, dado que responde a un cierto conjunto interesante de mensajes. Un objeto es **polimórfico con** otro objeto cuando ambos responden a un mismo mensaje (de formas distintas). El mensaje en cuestión es **polimórfico** y permite que

ORIGEN DE LA IDEA DE OBJETOS Y MENSAJES

Estos conceptos están basados en la biología celular, donde las células poseen una **frontera** llamada pared celular, la cual oculta y protege el interior de éstas que se comunican entre sí por medio de **mensajes** químicos. En la pared celular se encuentran los **receptores,** que son proteínas que le permiten recibir los mensajes químicos. ¡Dynamic dispatch en la naturaleza!

haya polimorfismo entre los dos (o más) objetos. Es posible desde el momento que se desacopla el envío de mensajes, cuando se produce la recepción de uno de ellos y viendo la forma en que actúa el objeto receptor en consecuencia. Que la decisión de lo que se ejecuta cuando se recibe un mensaje sea independiente del envío de este, se conoce como **dynamic dispatch** (despacho dinámico).

Tipos y Subtipos

El concepto de **tipo** está estrechamente ligado al paradigma de objetos, por más que no sea exclusivo a él (proviene del mundo lógico-matemático). Un **tipo** es un concepto que define un conjunto de objetos que comparten ciertas características (responden a un mismo conjunto de mensajes) que tienen sentido en un determinado contexto. Esto permite razonar sobre el conjunto, y todo lo que aplica a él aplica a cada integrante de este. Por ejemplo, si decimos que el tipo **Número** representa a todos los objetos que saben responder a la suma y la resta, entonces los objetos 1, ½ o Pi al ser números pueden ser usados en cualquier lugar que se necesite un número, ya que se pueden sumar y restar.

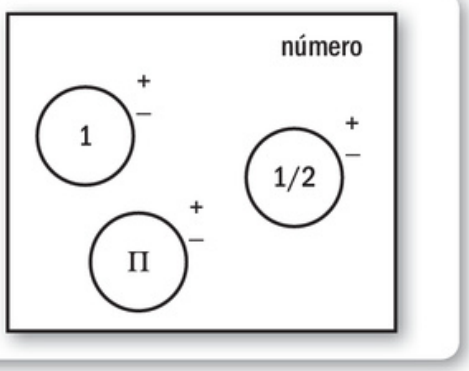


Figura 5. En esta imagen podemos ver al tipo denominado *Número* que se encarga de englobar otros números.

Ahora, el número 1 es un entero, mientras que el número ½ es una fracción (racional) y Pi es uno real (irracional). Todos son números, pero cada uno pertenece a un tipo especial, dado que tienen características distintas. Diremos entonces, que Entero, Fracción y Real son **subtipos** de Número, dado que pertenecen a este grupo (se suman y restan) pero también tienen diferencias que forman un subconjunto dentro del conjunto más grande. Por ejemplo, una fracción puede

responder a mensajes como numerador y denominador, pero un irracional no. Se dice que una Fracción es una **especialización** de Número. Así, debemos tener en cuenta que Número es un **supertipo** de Fracción y a su vez, Fracción un **subtipo** de Número. Entonces, Número, Entero, Fracción y Real forman una **jerarquía**.

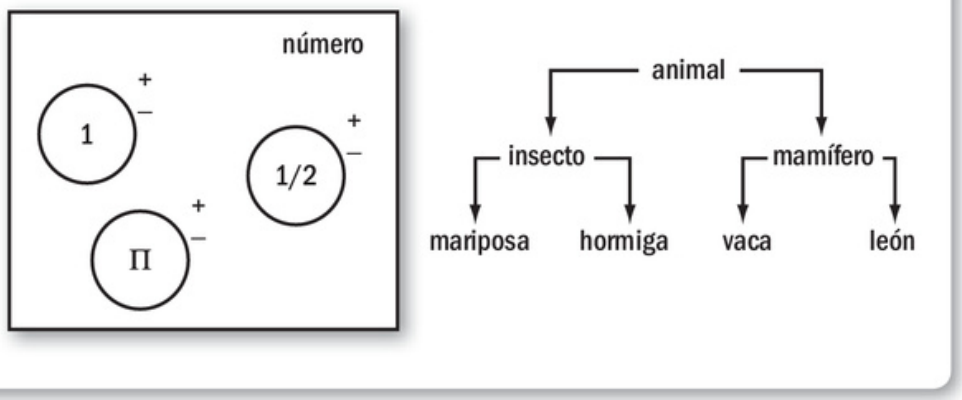


Figura 6. En esta imagen podemos ver los diagramas correspondientes a jerarquía de números y jerarquía de animales.

Un **supertipo** es más **abstracto** que un subtipo mientras que el **subtipo** es más **concreto** (especializado) que el **supertipo**. **Abstraer** significa que intencionalmente se dejan de lado ciertas características para poder así enfocarse en otras más importantes (dependiendo del contexto) en un determinado momento. Por lo tanto es simple ver que para poder abstraer es necesario primero contar con las instancias concretas. Luego se realiza el trabajo intelectual de seleccionar las características comunes y dejar de lado las diferencias.

¿CÓMO ABSTRAER?

No deberíamos empezar pensando la abstracción y luego las especializaciones. Ese proceso no funcionará porque seleccionaremos características que no son comunes o no seleccionaremos algunas que sí lo sean. Es conveniente empezar por los casos concretos que nos guiarán en la construcción de una abstracción cuando distingamos las características comunes entre ellos.

Herencia

En la programación orientada a objetos se considera que una **clase hereda** de otra cuando la primera es un tipo de la segunda. Una clase es un artefacto de implementación de tipo que permiten los lenguajes. Las clases forman jerarquías de herencia y se dice que cuando la clase Mamífero hereda de la clase Animal, Mamífero es una subclase de Animal y Animal es una **superclase** de Mamífero. En general, debemos saber que los lenguajes sólo permiten heredar de una sola clase aunque hay algunos, como el lenguaje de programación **C++,** que tienen herencia múltiple, o sea que una clase puede heredar de muchas otras.

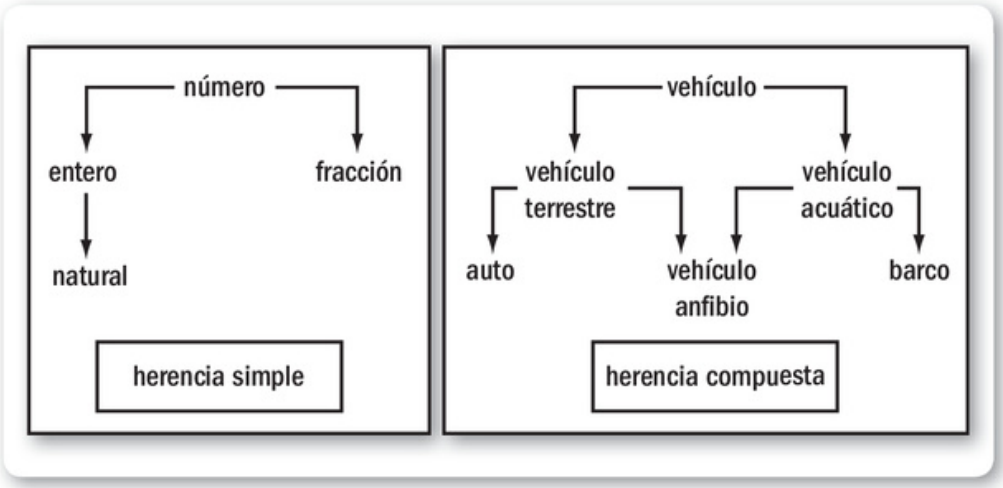


Figura 7. Jerarquía de herencia entre clases. Primero un ejemplo de una jerarquía con herencia simple, luego otro donde hay herencia múltiple. En general la herencia múltiple resulta en jerarquías más rebuscadas.

La **subclasificación** se parece al **subtipado** aunque no hay que confundirse, la **subclasificación** no fuerza al **subtipado**. Podríamos tener una clase que hereda de otra, pero sin compartir entre ellas un concepto de tipo para crear una jerarquía de tipos. Veamos un ejemplo para que quede más claro. Supongamos que tenemos la clase que representa a las listas que responden a los mensajes **¿tamaño?**, **elemento en la posición X** y **agregar al final**; y queremos tener la clase **Pila** (imaginemos una pila de platos), donde sólo se puede **poner arriba** y **sacar el de arriba**. Si quisiéramos implementar dicha clase, podríamos

utilizar la clase **lista** que ya tenemos y reutilizarla. Si heredamos de lista, y hacemos que **poner arriba** use **agregar al final,** y **sacar el de arriba** use **elemento en la posición X** más **¿tamaño?,** ya tendríamos todo resuelto.

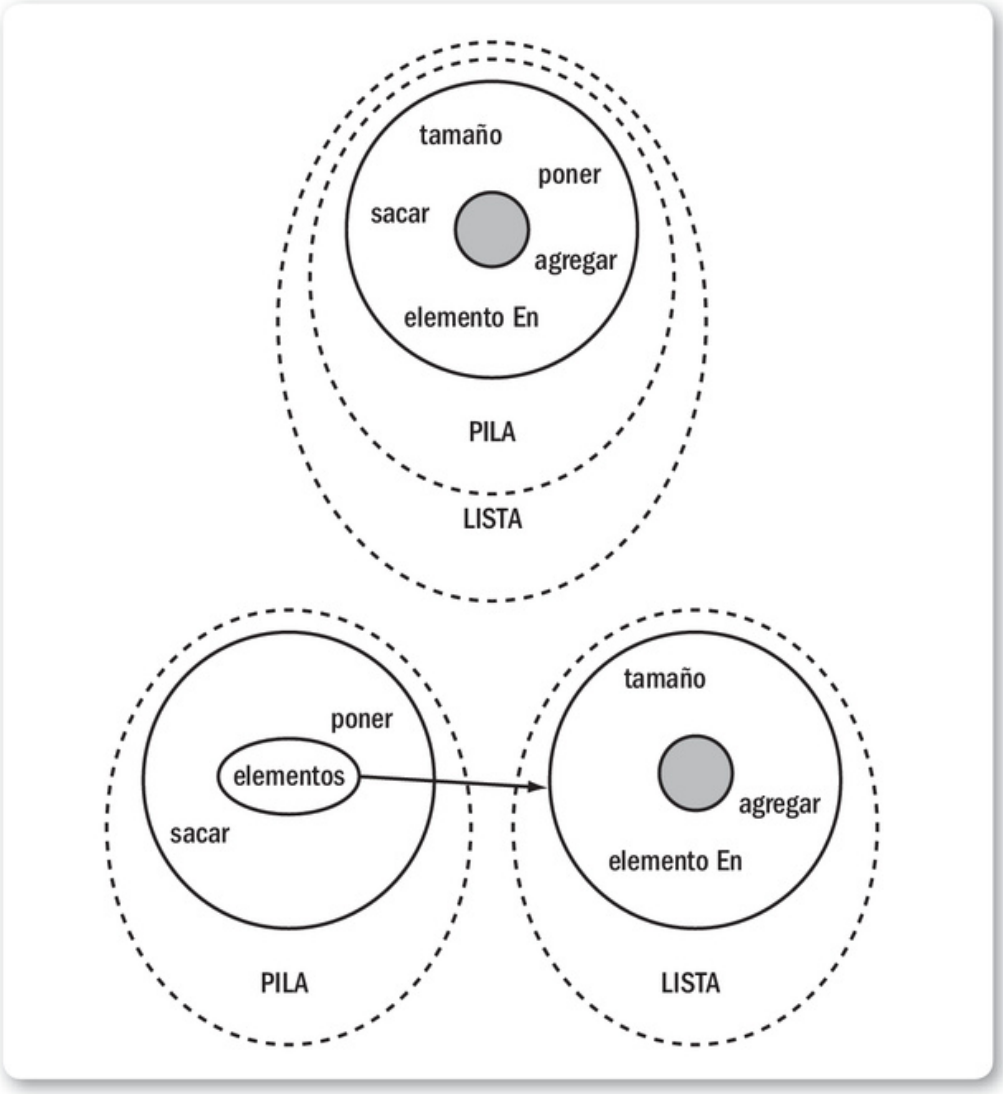


Figura 8. Herencia entre Lista y Pila. Observamos las dos alternativas para implementar una pila, la herencia y la agregación.

Lamentablemente el resultado no sería una relación de **subtipado** entre la lista y la pila, dado que la pila también sería una lista (podría responder a **elemento en la posición X**), cosa que no correspondería con el concepto de pila. Es claro que una lista y una pila no forman una relación de tipo, una pila no es una lista.

En general se considera mala práctica utilizar la **subclasificación** para **reutilizar** implementación sin formar una relación de **subtipado**. Generalmente la técnica que se utiliza para no crear ese tipo de jerarquías mal aplicadas es la **agregación**.

Debemos tener en cuenta que se llama agregación cuando un objeto utiliza a otro. Se dice que un objeto **tiene** o **usa** otro.

Veamos otro ejemplo: supongamos que tenemos Rectángulos y Cuadrados, y queremos que respondan al mensaje **¿área?**. Ambos tipos son muy parecidos, podríamos decir que un Cuadrado es un Rectángulo con los lados iguales o también que un Rectángulo es un Cuadrado con los lados desparejos. Podríamos, entonces, elegir alguna de estas dos jerarquías para reutilizar el comportamiento. ¿Cuál conviene elegir?

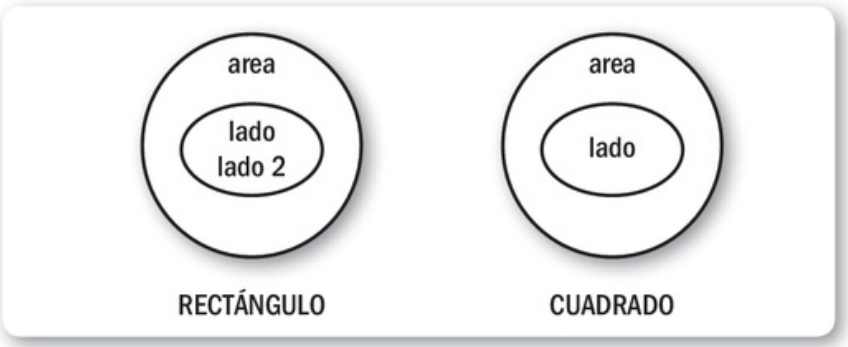


Figura 9. Objetos que representan a un rectángulo y a un cuadrado. El rectángulo especializa al cuadrado agregando un colaborador extra para representar el lado distinto.

Viendo las diferencias entre los objetos, elegir heredar Rectángulo de Cuadrado parece una opción sencilla, solo hay que agregar un

JERARQUÍA MAL UTILIZADA

En su mayoría, si una clase hereda métodos que no tienen ninguna relación con ella, que no aplican y que no forman parte de las **responsabilidades** que debería tener, entonces estamos frente a un mal uso de la herencia. Debemos evitar este tipo de jerarquías dado que generan productos de mala calidad. La forma correcta de aplicarlas es eliminando la herencia y utilizando la agregación.

colaborador interno **lado2** y cambiar el comportamiento del mensaje **¿área?**. Detengámonos un momento a pensar, imaginemos que necesitamos un Cuadrado, y que al ser el Rectángulo subclase de Cuadrado, todo rectángulo es un cuadrado, de esta forma podríamos usar un rectángulo en su lugar. ¿Suena lógico esto? Claramente no.

Veamos la otra opción, si Cuadrado hereda de Rectángulo, usaríamos sólo un lado y modificaríamos **¿área?,** o podríamos hacer que ambos lados sean iguales y no tendríamos que modificar nada. Ahora si necesitamos un Rectángulo y tenemos un Cuadrado, no hay problema, dado que un cuadrado es un rectángulo válido. Reutilizamos este conocimiento y la jerarquía estará basada en una relación de tipos.

Recordemos: siempre debemos preocuparnos de hacer que nuestras jerarquías estén basadas en una relación de tipos.

En el ejemplo anterior dijimos que un Cuadrado podía ir en lugar de un Rectángulo. Formalmente quisimos decir que el Cuadrado podía **sustituir** a un Rectángulo o, dicho de otra forma, que un Rectángulo es **sustituible** por un Cuadrado. La herencia fomenta el polimorfismo, porque generalmente los lenguajes tienen ciertas reglas que fuerzan a que las subclases puedan sustituir a las superclases. ¿Qué significa sustituir? Significa que, cuando un objeto que usamos es **sustituido** por otro, esperamos que este objeto nuevo nos requiera lo mismo o menos y que nos dé lo mismo o más que el objeto que sustituyó. De esta forma nuestras **expectativas** y nuestras **colaboraciones** con este objeto no se verán afectadas.

Herencia simple versus herencia múltiple

Si bien la mayoría de los lenguajes orientados a objetos que se basan en las clases utilizan herencia simple, hay algunos otros que permiten heredar de muchas clases al mismo tiempo. El caso más famoso es el lenguaje **C++**. ¿Por qué existe la herencia múltiple? La herencia múltiple permite que las clases pertenezcan a distintas jerarquías adquiriendo el conocimiento, o sea reutilizando código, de distintas clases. A primera vista nos puede parecer que es una ventaja contra la herencia simple, pero generalmente pasa que las jerarquías se deforman, y dejan de estar guiadas por relaciones de tipos para

pasar a estar guiadas por la reutilización de código. De esta forma, en su mayoría terminan con complejas e inutilizables clases. Además la herencia múltiple plantea ciertos problemas de implementación que se desencadenan en errores molestos de corregir. El principal problema es en qué orden se hereda, es decir, qué clase tiene prioridad sobre otra, lo cual lleva a que algunas clases oculten o sobrescriban los métodos de otras. Es difícil que el autor de una jerarquía haya preparado el diseño de esta para actuar junto con otra totalmente dispar, creada por otra persona. La gente que trabaja con este tipo de lenguajes es muy cuidadosa a la hora de utilizar la herencia múltiple y en lo posible tratan de evitarla. Por eso y por algunos otros problemas, la mayoría de los lenguajes opta por la herencia simple (sumada a otros mecanismos) para la creación de jerarquías de tipos.

> LA MAYORIA DE LOS LENGUAJES OPTA POR LA HERENCIA SIMPLE

¿Por qué objetos?

Cuando nos encontramos frente a un problema o una situación que queremos entender, para luego resolver, creamos en nuestra mente un modelo que lo representa. Un modelo no refleja totalmente la problemática. Nosotros mismos, al crear el modelo, elegimos enfocarnos en algunas características y dejamos otras de lado. Luego, operamos sobre este y contraponemos los resultados con el problema real. Es necesario tener en cuenta que, en base a las conclusiones que nos encargamos de obtener, volvemos al modelo y lo ajustamos para lograr mejores resultados la próxima vez.



ALTERNATIVAS A LA HERENCIA MÚLTIPLE

Para no restringir el **tipado** a la herencia simple, subclasificando innecesariamente, se usan las **interfaces** (**Java**, las veremos más adelante), los **mixins** (**Ruby**) o los **traits** (**Smalltalk**). Cada uno tiene ventajas y desventajas, respecto de la herencia múltiple, que permiten **reutilizar** y el **tipado**.

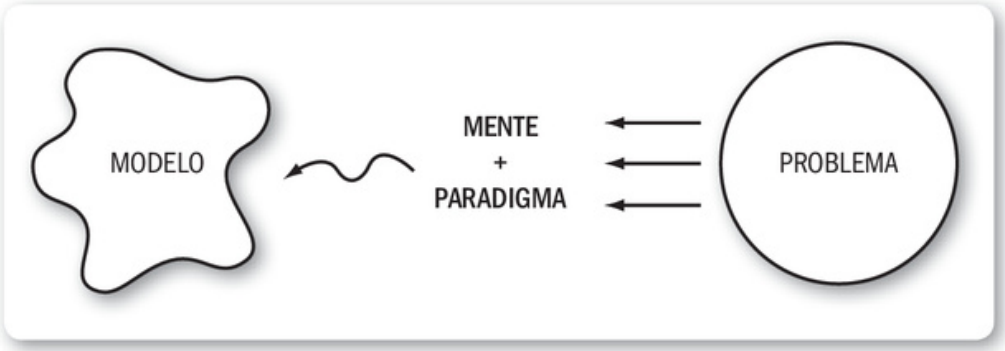


Figura 10. El ciclo de modelado y cómo interviene el paradigma. De toda la información que da el problema real, parte se deja de lado explícitamente y se focaliza en otra.

El paradigma de objetos es un filtro que se encuentra entre el problema y el modelo, que polariza el entendimiento de la situación acomodándolo a sus conceptos. Afortunadamente, el paradigma de la orientación a objetos permite que la interpretación entre la realidad, la problemática y el modelo sea muy fácil. Un objeto es tangible y es fácilmente relacionado a un elemento del dominio del problema, la carga mental que implica el pasaje del modelo al problema es imperceptible. Esta es la gran ventaja frente a otros paradigmas de programación, dado que permite interpretar fácilmente la problemática. Además la comunicación entre individuos que se manejen tanto en el modelo (programadores) como en la situación de interés (clientes) es sencilla, ya que se maneja el mismo idioma en ambos mundos. Se dice que entre el modelo y el problema debe existir una relación de isomorfismo.

Un isomorfismo es una relación entre dos elementos cuando ambos son **indistinguibles**, ambos cumplen las mismas propiedades y todo lo que es cierto para uno lo es para el otro, del mismo modo todo lo que es falso para uno lo es para el otro. Se dice que ambos elementos son **isomorfos** entre sí. Por ejemplo, si estamos en el ámbito bancario, tendremos un objeto Cuenta que representa una cuenta real en el banco. Entonces cuando el cliente diga **cuenta** nosotros entenderemos que se trata de un objeto de este tipo en forma inmediata. Y viceversa, cuando nosotros hablemos de una Cuenta, el cliente comprenderá, sin ningún inconveniente, que queremos decirle algo sobre una de ellas. De este ejemplo podemos sacar algunas conclusiones; no puede haber

un elemento del modelo que represente a más de un concepto del dominio por modelar. Tampoco lo opuesto, que un concepto del problema esté representado por más de un elemento del modelo.

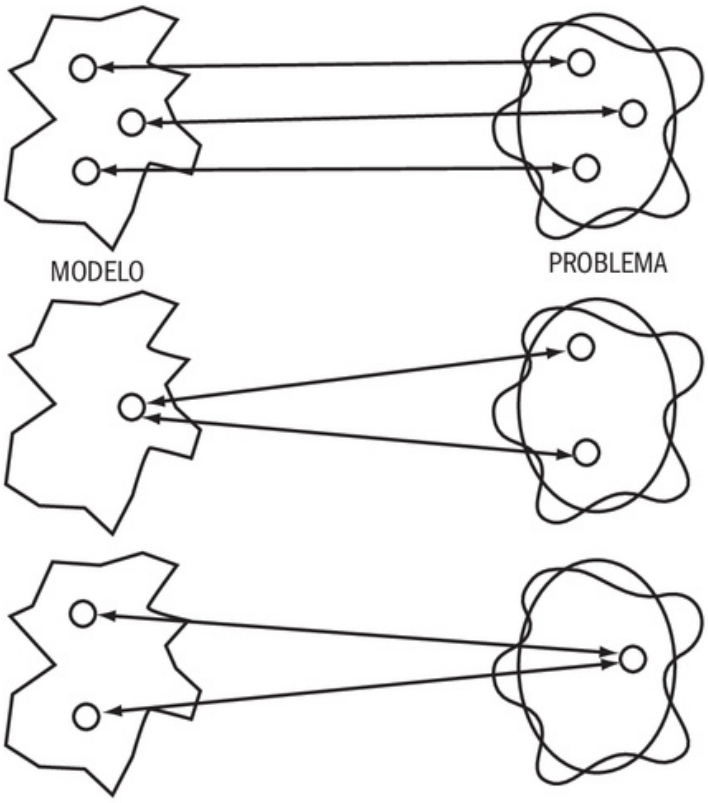


Figura 11. En esta imagen vemos el isomorfismo entre el modelo y el problema (primer ejemplo). Así como también los casos que debemos evitar (los siguientes).

RESUMEN

La programación orientada a objetos se basa en que todo es un objeto y en que estos se comunican mediante mensajes. Tiene poderosas ideas como el tipado y el polimorfismo. La mayoría de la los lenguajes utilizan las clases para tipar y la herencia para subtipar. Existen lenguajes con herencia simple y otros con herencia múltiple, aunque también se utilizan otras alternativas para flexibilizar las jerarquías. Es importante entablar un isomorfismo entre los elementos del dominio y los elementos del modelo que realizamos.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Cuáles son los conceptos básicos del paradigma?
2	¿Qué es el encapsulamiento y el ocultamiento de la información?
3	¿Qué beneficios traen el encapsulamiento y el ocultamiento de la información?
4	¿Para qué sirven los tipos?
5	¿Qué es el polimorfismo?
6	Defina herencia.
7	Diferencias entre el tipado y la clasificación.
8	¿Para qué se utiliza la agregación?
9	¿Qué condición debe cumplir un modelo para ser una buena representación?
10	¿Por qué es necesario utilizar objetos?

ACTIVIDADES PRÁCTICAS

1	Ingrese a la dirección **www.smalltalk.org/smalltalk/history.html** y lea "The Early History of Smalltalk" de Alan Kay (sólo en inglés).
2	Modele un átomo.
3	Diseñe una molécula como subclase de Átomo.
4	Especifique una molécula como una agregación de átomos.
5	Relacione los tipos Animal, Mamífero, Ave, Pingüino, Cóndor y Elefante en relaciones de subtipos.

*02 –

Iniciación a Java

Desde sus orígenes Java se ha difundido en diversos ámbitos con el afán de consagrarse como la gran plataforma que es hoy. Antes de introducirnos por completo en la programación Java necesitamos conocer su historia y las fuerzas que guiaron su creación. Veremos también una introducción a la forma de trabajo que adoptaremos a lo largo del libro, así como un breve vistazo a la técnica de Test Driven Development en Java.

- **Historia** 30
- **Preparación** 32
  - Eclipse IDE 33
  - Test Driven Development 36
- **Primeros códigos** 38
- **Resumen** 43
- **Actividades** 44

Historia

El lenguaje de programación Java tiene sus orígenes en el año 1991, cuando **Sun** empieza el proyecto **Green**. Este proyecto tenía como objeto controlar dispositivos hogareños, para lo que crearon un lenguaje llamado **Oak**. Sun no tuvo éxito, y no es hasta 1995 cuando el nombre es cambiado a Java y se lanza al mundo.

Inicialmente, Java se lanzó como un lenguaje cuyos programas podían correr en cualquier plataforma. El slogan de Sun era **write once, run anywhere** (escribir una vez, correrlo en cualquier parte). Para lograr esto Java corre sobre una **máquina virtual** o un programa que simula una máquina abstracta, la cual funciona aislando al programa que corre sobre ella de los distintos hardwares y sistemas operativos. De esta forma, para el programa que estamos utilizando, la maquina donde corre es siempre igual.

James Gosling, padre de Java, lo hizo parecido a **C++** para que los programadores de este lenguaje se sintieran cómodos con Java y optaran por él. Java ofrecía a los programadores un lenguaje parecido a C++ pero simplificado (por ese entonces Java no poseía genéricos) y tenía manejo automático de memoria, es decir, que el programador no es el responsable de liberar la memoria que no usa. De esto se encarga una función de la máquina virtual llamada recolector de basura (garbage collector). El **garbage collector** es un proceso que se ejecuta paralelamente al de la aplicación y se encarga de liberar la memoria ocupada por los objetos que no son utilizados por el sistema.

Java siempre fue un lenguaje que triunfó gracias a Internet. En un comienzo lo hizo a través de las applets, pequeñas aplicaciones embebidas en las páginas web que se ejecutan en los navegadores. En ese entonces las applets no triunfaron dado que competían con la tecnología **Shockwave**, ahora conocida como **Flash**. A partir de 1997,



LÍNEA DE TIEMPO DE JAVA

En **www.java.com/en/javahistory/timeline.jsp** podemos ver una línea temporal de los eventos importantes en la historia de Java y de aquellos del mundo tecnológico que los acompañaron. La rápida evolución del lenguaje y de las distintas tecnologías que se crearon a partir de él son evidentes.

Java empezó a migrar hacia los servidores. Se lanzó la primera versión de los **Servlets**, componentes Java que corren en los servidores. En esta área es donde este lenguaje se impuso y brilló. Las tecnologías competidoras de ese entonces eran **CGI**, **ASP** y **PHP**. Java superó a la competencia y se posicionó como la tecnología por excelencia para las aplicaciones web empresariales. Desde entonces Java es una de las tecnologías más importantes, disponible en casi todas las plataformas. Se encuentra en las desktops (aplicaciones de escritorio), el los servidores (sirviendo páginas web y aplicaciones web) y hasta en los teléfonos celulares (versión micro de Java).

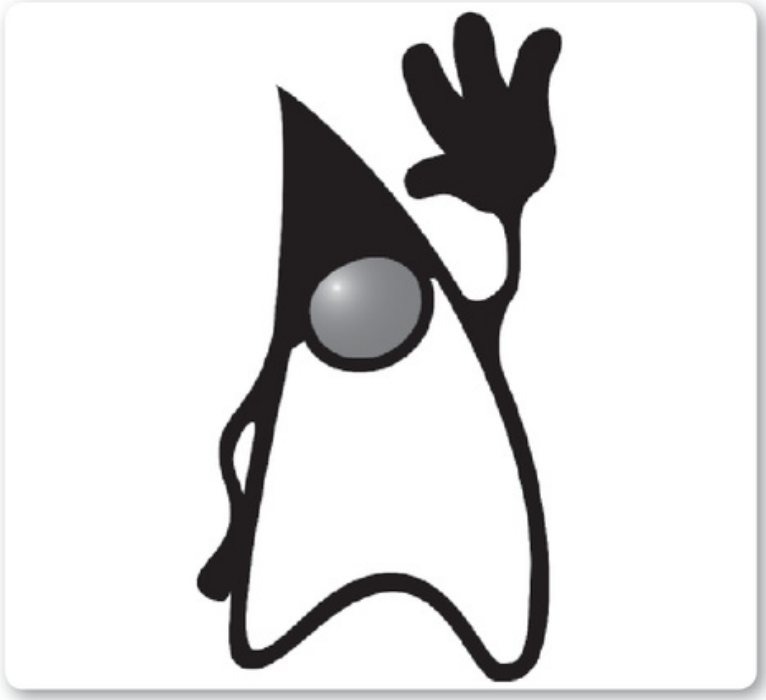


Figura 1. Duke, la mascota de Java en los comienzos del lenguaje, fue muy popular y atrayente. Se lo podía ver en cada artículo, libro o sitio relacionado con este lenguaje.

Ahora Java forma parte de uno de los sistemas operativos para celular más famosos y pujantes que hay en la actualidad: Android. Este sistema operativo fue creado por los desarrolladores del famoso buscador **Google**; está basado en Java, por lo cual todas las aplicaciones para los celulares con Android están hechas utilizando este lenguaje. No sólo existen celulares con el sistema operativo

Android, sino smartphones, netbooks y tablets que también lo utilizan. En el 2009 Oracle, gigante de las bases de datos, compró Sun y, por lo tanto, también asumió la responsabilidad sobre la evolución de Java.

El futuro de Java es prometedor, ya que esta respaldado por gigantes de la informática y por una enorme comunidad de programadores.



Preparación

Para poder programar en Java, compilar y correr los programas, necesitaremos instalar el kit de desarrollo. Podemos conseguirlo en **www.oracle.com/technetwork/indexes/downloads/index.html**.

Debemos asegurarnos de descargar la última versión del SDK, Software Development Kit (Kit de Desarrollo de Software), y no el JRE, Java Runtime Environment (Entorno de Ejecución Java). El primero nos permitirá crear y ejecutar aplicaciones Java, el segundo solamente correrlas. El ejecutable que descargaremos nos guiará durante la sencilla instalación. Finalizada esta tendremos a nuestra disposición los distintos programas para producir y ejecutar aplicaciones Java.

El compilador correspondiente a este lenguaje: **javac.exe** (el cual se encarga de producir los binarios Java a partir del código fuente), el intérprete **java.exe** (que ejecuta un programa Java) y el documentador **javadoc.exe** (el que se encargará de generar la documentación necesaria para las clases) entre otros. También encontraremos el **profiler jvisualvm.exe**, una herramienta que nos permitirá medir la **performace** (tiempo de ejecución y uso de memoria) de cualquier programa Java de una forma sencilla y visual. Así mismo contaremos con las librerías base que trae Java por defecto y el código fuente de estas, por si queremos ver cómo funcionan internamente.



TUTORIALES Y EJEMPLOS

Tanto en la página donde descargamos el SDK cómo dentro de él, encontraremos ejemplos y tutoriales que nos ayudarán a profundizar los conocimientos que incorporaremos a lo largo del libro. Vale la pena mirar los ejemplos incluidos y probarlos, ya que utilizan muchas clases importantes de la librería base.

Eclipse IDE

Ya estamos en condiciones de empezar a programar nuestros primeros códigos en Java. Podríamos comenzar escribiendo con cualquier editor de texto y luego utilizar el compilador, programa que toma un archivo con código fuente, lo transforma y genera otro que entiende la **virtual machine** (más conocida como máquina virtual o también VM). Pero esta forma de trabajar solamente sirve para algunas pruebas sencillas, no sólo es apta para realizar desarrollos profesionales. Por lo tanto también necesitaremos descargar un entorno de trabajo. Utilizaremos el IDE, Integrated Development Environment (Entorno de Desarrollo Integrado) llamado Eclipse.

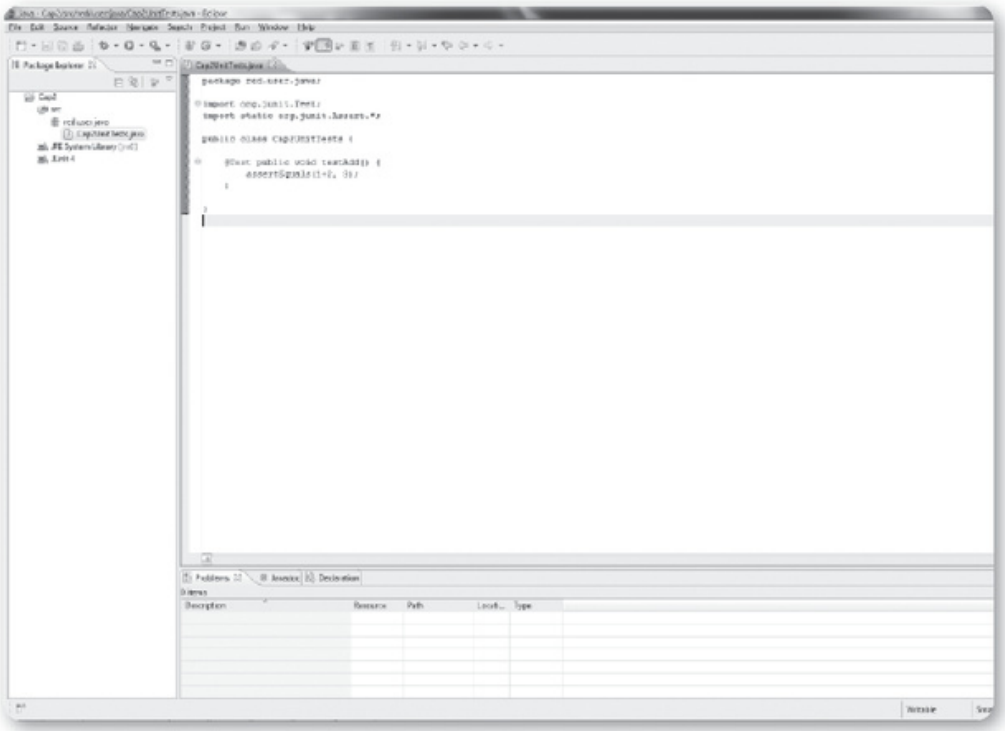


Figura 2. En esta imagen podemos ver la vista de Java, se trata de la vista principal de Eclipse y es en la que más se trabaja.

En **www.eclipse.org** encontraremos los enlaces para descargar el entorno de desarrollo más famoso. Hallaremos los distintos **sabores** de Eclipse para poder desarrollar en diversas tecnologías desde Java, C++ y PHP hasta Javascript. Allí también encontraremos diversas herramientas para agregarle a nuestro IDE.

Eclipse es el entorno de desarrollo Java por excelencia. Fue concebido como una plataforma para la creación de IDEs, cuya expansión puede ser realizada mediante plugins. Inicialmente los lenguajes soportados eran Java y luego C++. Actualmente existe una amplia variedad de plugins para casi todos los lenguajes, y los lenguajes nuevos generalmente utilizan Eclipse dado que provee la infraestructura básica para la creación del IDE que requieren. Nosotros descargaremos la versión para Java Developers. El archivo posee una extensión .ZIP (archivo comprimido), que descomprimiremos en el lugar que queramos. Luego, no se requiere más instalación y, una vez descomprimida la carpeta, ejecutaremos el archivo llamado **eclipse.exe** (nos conviene crear un acceso directo desde el escritorio para futuros usos).

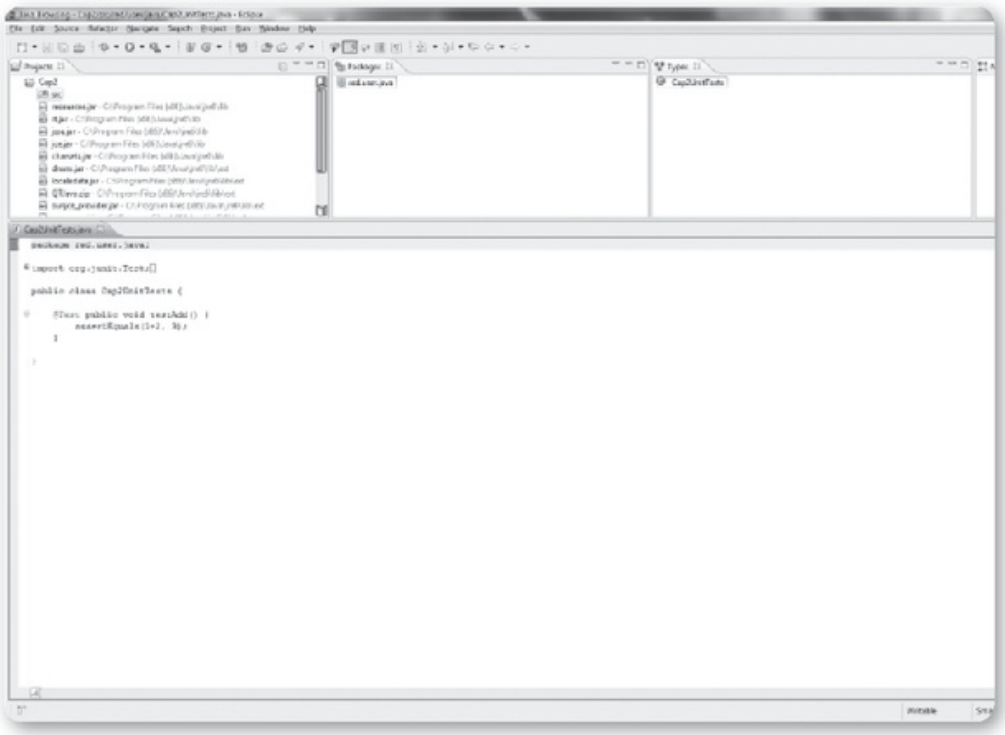


Figura 3. Vista de **`Java Browsing`** focaliza la atención en el área de edición y en la navegación de paquetes, clases y métodos.

Lo primero que veremos será una ventana que nos preguntará donde queremos guardar el **workspace** (área de trabajo). El workspace es un directorio donde Eclipse salvará los distintos proyectos en los que trabajemos. La pantalla de bienvenida nos recibe con ejemplos y

tutoriales, así como también con una visita a las características que ofrece la herramienta. Si elegimos ir al **workbench** (mesa de trabajo) nos encontraremos con la vista Java. Eclipse tiene el concepto de vistas para trabajar, cada vista es una configuración de paneles y ventanas que permiten al usuario enfocarse en una determinada tarea.

Para trabajar con Java tenemos tres vistas principales, la **vista de Java**, **la vista de Java Browsing** y **la vista de Debug**.

En la vista de Java encontramos un explorador de paquetes (Pakage Explorer), fuentes Java y librerías del proyecto. Incluye también el panel de errores (Problems), donde Eclipse nos informará de aquellos relacionados con el proyecto (librerías que faltan, errores sintácticos en los códigos fuente, etc.). El panel del esquema (Outline) de las clases y paquetes también está presente. Podemos darnos cuenta de que la parte central de la vista está destinada a la edición de código.

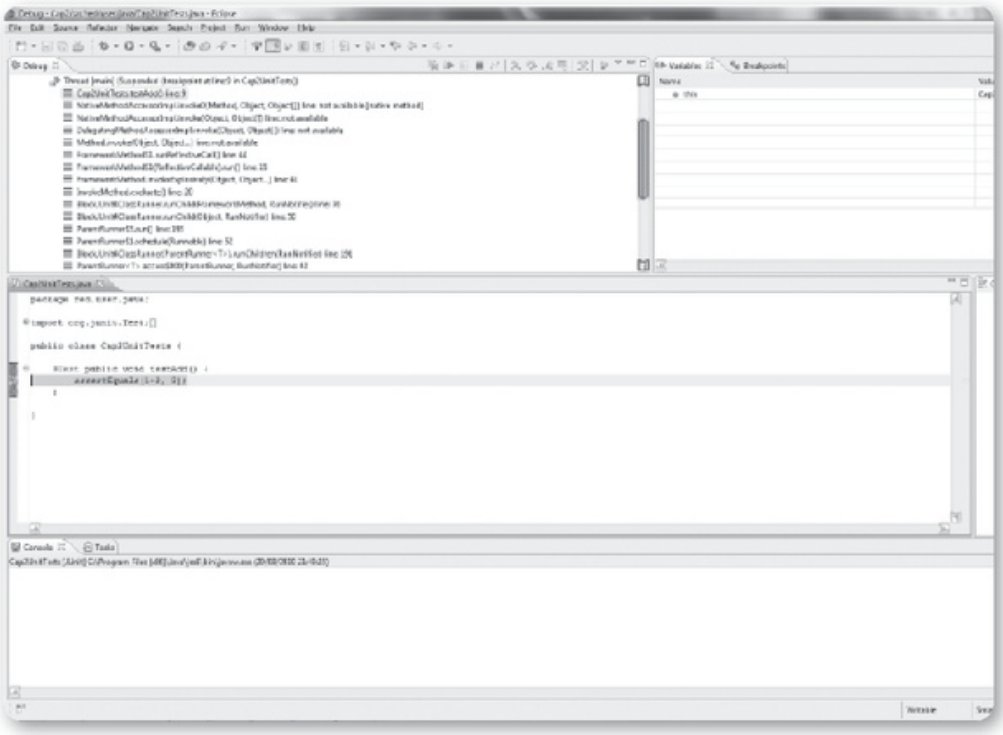


Figura 4. La vista de *Debug* permite que seamos testigos de qué ocurre durante la ejecución de nuestro código.

La segunda vista, la de Java Browsing, está enfocada en el trabajo con las clases y nos recuerda al entorno de trabajo clásico de Smalltalk. En

esta vista nos encontramos con paneles en la parte superior, y el resto del espacio es destinado a la edición de los códigos fuente. Los cuatro paneles son: el de proyecto (Proyects), el de paquete (Packages), el de clases (Types, que presenta las distintas clases dentro del paquete) y el de miembros (Members, que se encarga de mostrar los elementos que conforman la clase seleccionada en el panel anterior).

> USAR ECLIPSE AGILIZA MUCHA EL DESARROLLO YA QUE INTEGRA MUCHAS HERRAMIENTAS

Finalmente, la vista de Debug, se activa cuando la ejecución (lanzada en modo Debug) alcanza un breakpoint (punto de frenado) en el código. En ella podemos hacer avanzar la ejecución, detenerla, inspeccionar qué objetos están siendo usados en un determinado momento y hasta modificarlos. Debemos tener en cuenta que esta es la vista que más usaremos cuando queramos determinar en forma exacta qué está haciendo el código y por qué no hace lo que nosotros queremos.

Test Driven Development

A lo largo del libro trabajaremos utilizando JUnit. JUnit es un herramienta (que usaremos a través de Eclipse) para trabajar aplicando una metodología conocida como **Test Driven Development** (**TDD**) o Desarrollo Guiado por Pruebas. TDD es un estilo para plantear el desarrollo de un software, que se basa en el hecho de que el programador vuelca su conocimiento sobre determinado problema en forma de aserciones. El programador debe codificar lo necesario para validar estas aserciones que enunció anteriormente. Cuando todas sean



AUTOCOMPLETE EN ECLIPSE

Si estamos posicionados dentro de un método y presionamos las teclas **CTRL** y la **BARRA ESPACIADORA**, se accionará el autocomplete. Esta funcionalidad, dependiendo del contexto, nos muestra qué elementos tenemos a disposición, ya sean clases, atributos o métodos (de instancia y estáticos). También nos da más información sobre los métodos, como su tipo de retorno y los parámetros que acepta.

válidas debe seguir explorando el dominio del problema agregando más aserciones. Cuando alguna resulte falsa, debe codificar para hacerla verdadera. Finalmente, el programador refactoriza el código y lo repara buscando abstracciones, de forma de simplificarlo pero procurando que el comportamiento no cambie; las aserciones deben continuar siendo válidas.

> LOS TESTS FORMAN UNA RED DE SEGURIDAD AL MOMENTO DE CAMBIAR EL CÓDIGO

TDD ofrece una forma de abordar el problema de manera exploratoria, usando ejemplos (aserciones). Esto es más fácil que tratar de resolver el problema en abstracto y para todos los casos. Así mismo TDD hace que el programador sea el usuario de su propio código primero, logrando que él mismo pueda darse cuenta de cuán amigable y fácil de usar es el API desarrollado, también facilita el mantenimiento del software. El software, por su naturaleza, está en constante cambio. Estos cambios tienen que ser codificados e introducidos en el código existente sin afectar involuntariamente al resto de las funcionalidades del sistema. Así, los test sirven como red de seguridad, dado que podremos verificar rápida y sistemáticamente el correcto funcionamiento de la aplicación.

JUnit es la herramienta clásica, originaria de Smalltalk, para realizar TDD en Java (otra conocida es TestNG). Aparece de la mano de **Kent Beck**, el creador del **Extreme Programming** (Programación Extrema), y luego es migrada a Java y a casi todo lenguaje existente.

En JUnit se definen casos de prueba, representados por una clase que contiene métodos de prueba. Cada uno de estos métodos tiene el objetivo de probar uno o algunos ejemplos y aserciones. JUnit luego ejecuta los casos y genera un reporte que informa cuáles pasaron exitosamente y cuáles fallaron o dieron error. En Eclipse contamos con un plugin ya instalado que nos permite utilizar JUnit con unos pocos clics.



BDD O BEHAVIOR DRIVEN DEVELOPMENT

BDD (desarrollo guiado por comportamiento) es una extensión a TDD que hace foco en escribir los test en un lenguaje cercano al natural, con las especificaciones obtenidas del cliente y las que este puede entender. De esta forma, la brecha entre el programador y el cliente se reduce, ya que comparten un mismo léxico.

Primeros códigos

Vamos a trabajar en nuestros primeros códigos en Java. Para eso necesitamos crear un **proyecto** donde escribirlos. Será necesario que iniciemos Eclipse y posteriormente hacer clic en el menú **File** o sobre el botón **New** que se encuentra en la barra de herramientas. También podemos hacer clic con el botón derecho del mouse en el **Package Explorer** y seleccionar la opción **New.../Java Project**.

Debemos elegir un nombre para el proyecto y escribirlo en la ventana adecuada, luego presionar sobre **Finish**.



Figura 5. Pantalla de creación de proyectos Java. De forma sencilla a través de las opciones que ofrece el asistente, podemos controlar cada parámetro de la creación.

Con esto conseguiremos tener un proyecto y dentro de él la carpeta **src** que será donde alojaremos los archivos fuentes. En forma predeterminada Eclipse agrega como dependencia las librerías que vienen con la instalación de Java. El paso siguiente es crear nuestro primer caso de prueba (**test case**). En Java todo código tiene que estar contenido en una clase, nuestro test case no será la excepción.

Para crear un test case, en el menú **File** seleccionamos **New.../JUnit Test Case**, elejimos la versión de JUnit que usaremos (JUnit 4), un nombre de package (**red.user.java**) y un nombre para el Test Case (**UnitTests**). Luego presionamos el botón **Finish**.

Eclipse alertará que no tiene la librería incluida y preguntará si queremos agregarla, seleccionamos la opción **OK**.

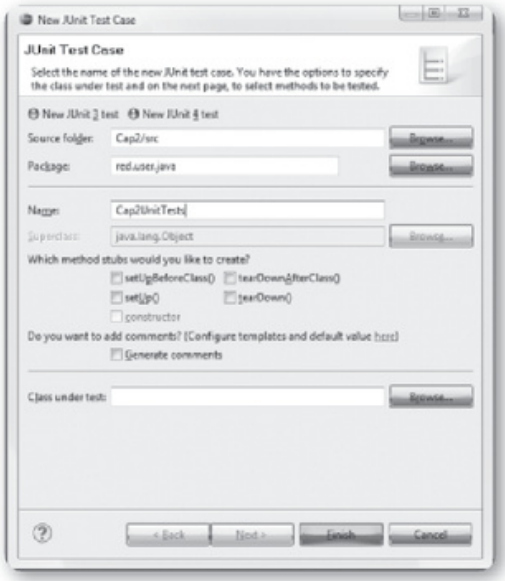


Figura 6. La pantalla de creación de test cases nos permite elegir la versión de JUnit con la que queremos trabajar y la creación de métodos auxiliares para los test.

Tenemos ya nuestra primera clase, que es un test case. Ahora vamos a hacer una simple prueba, veamos si efectivamente cuando sumamos uno más dos obtenemos tres. Para esto debemos escribir el siguiente código:

```
package red.user.java;
import org.junit.Test;
import static org.junit.Assert.*;
public class Cap2UnitTests {
}
        @Test public void testAdd() {
                assertEquals(1+2, 3);
        }
}
```

JUNIT

En **www.junit.org** podemos encontrar más información sobre JUnit, con documentación y ejemplos que nos ayudarán a mejorar la forma en que hacemos los test. La sección de preguntas y respuestas es un buen punto de partida para resolver todas las dudas que tengamos sobre los test. También podemos bajar el código fuente y aprender del funcionamiento interno de JUnit.

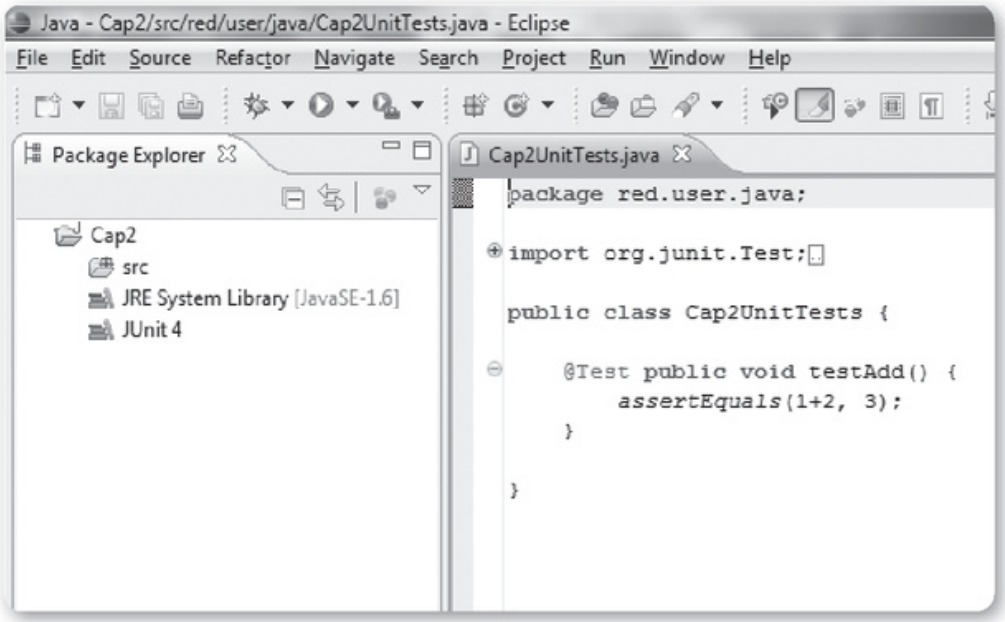


Figura 7. En esta imagen vemos como luce nuestro código en Eclipse, notemos los distintos estilos que utiliza el IDE para diferenciar los elementos que conforman el código escrito.

La palabra **class** seguida del nombre de la clase indica el inicio de esta. Luego tenemos el método, cuyo nombre es **testAdd**, que no devuelve ninguna respuesta (**void**) y que está marcado como un método de test (anotación **@Test**). En el cuerpo del método tenemos una aserción que dice: "¿Son iguales 3 y 1+2?". Para saber si es así, vamos a ejecutar el test y ver qué resulta.

Presionamos el botón derecho sobre la clase y seleccionamos **Run As.../JUnit Test**. Aparece entonces el llamado **runner** de JUnit que nos mostrará una barra verde si todo salió bien o roja si algo falló.

JAVA WORLD

Si nos dirigimos con nuestro navegador al sitio web que se encuentra en la dirección **www.javaworld.com** encontraremos abundante información sobre este lenguaje. Allí hallaremos las últimas noticias, artículos, tutoriales y también enlaces a distintos blogs que nos serán de gran utilidad y nos mantendrán actualizados sobre los temas más calientes del ambiente Java.

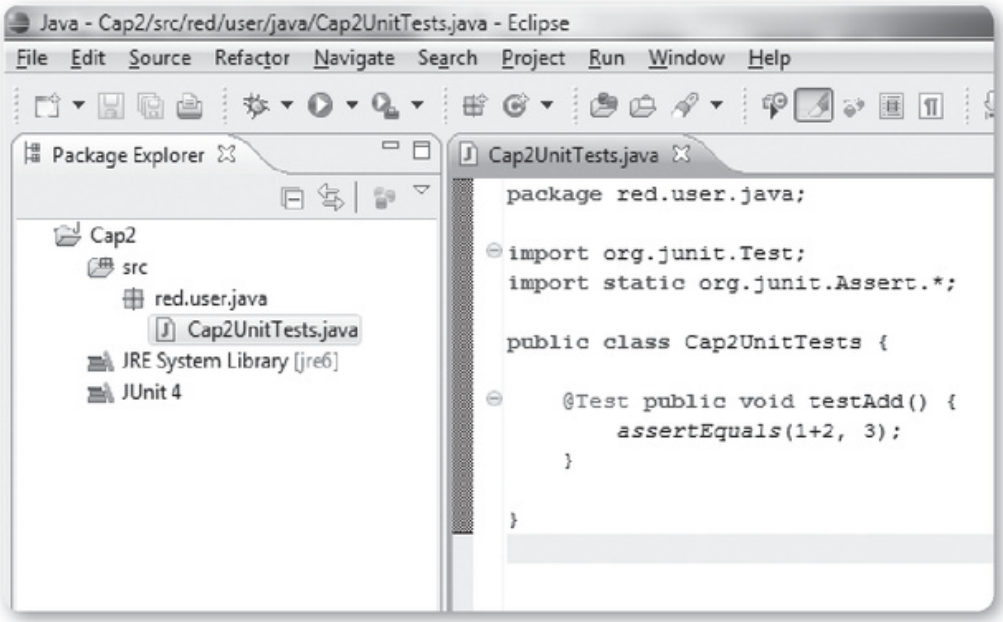


Figura 8. Al correr un test case el resultado se ve reflejado en el panel del runner de JUnit. Si la barra es de color verde, todos los test pasaron, si es roja, significa que alguno falló.

Ahora investiguemos con más profundidad qué hemos escrito y cómo es que funciona:

```
package red.user.java;
```

Todo archivo fuente Java define una clase, toda clase pertenece a un **paquete**. Un paquete es una forma de agrupar las clases en grupos que tengan cierto sentido. Se estila que el prefijo del paquete sea el dominio web de la organización a la que pertenece en orden inverso

GARBAGE COLLECTOR

Los lenguajes que no utilizan este mecanismo para administrar la memoria, como **C++**, requieren que el programador, manualmente libere la memoria que no se va a utilizar más. Esta tarea manual es la causante de muchos errores en los programas y del consumo de mucha memoria. Aquella que no se libera, pero tampoco se usa, se conoce como **memory leak**.

(ejemplo: com.google). En nuestro código definimos el paquete **red**, dentro de él el subpaquete **user** y dentro de éste el subpaquete **java**. Java fuerza a que cada paquete esté representado por un directorio en el sistema de archivos. En nuestro caso Eclipse automáticamente crea un directorio **red**, dentro de él otro llamado **user**, adentro uno llamado **java** y, finalmente, dentro de java el archivo **Cap2UnitTests.java**.

```
import org.junit.Test;
import static org.junit.Assert.*;
```

Estas sentencias indican que queremos **importar** la clase **Test**, localizada en el paquete **org.junit** y también todos los métodos estáticos (que pertenecen a la clase y no a las instancias de ella) de la clase **Assert**. En Java los métodos conocidos como **estáticos**, veremos más adelante, son distintos de los de instancias.

```
public class Cap2UnitTests
```

Indica que estamos definiendo la clase **Cap2UnitTests**. Todo lo que está entre la primera **{** y la última **}** corresponde con el cuerpo de la clase.

```
@Test public void testAdd()
```

Aquí definimos el método que ejecutarán las instancias de la clase **Cap2UnitTests** cuando se les envíe el mensaje **testAdd**. Primero tenemos la anotación **@Test** que indica que el método es un test. Las **anotaciones** son un artefacto de Java para agregar información sobre



VERSIONES ANTERIORES DE JUNIT

Es muy interesante tener en cuenta que hasta las versión 4, JUnit utilizaba la herencia para definir un test case basándose en la clase **TestCase**. Recordemos que dicha clase se encargaba de realizar la definición de los distintos métodos de aserción. Los métodos de test se reconocían dado que requerían obligatoriamente que tuvieran el prefijo "test". En las nuevas versiones de Junit esto se flexibilizó al hacer uso de las anotaciones, de esta forma tenemos más flexibilidad a la hora de trabajar.

los distintos elementos del lenguaje, llamamos a esta información **metadata**. Metadata significa datos sobre los datos, datos que dicen algo respecto de los datos. Meta significa hablar en un nivel superior. La anotación dice que el método es un test, no dice nada de lo que hace, de eso habla el método en sí. Luego el método esta marcado como público, lo que significa que puede ser invocado (o sea que se les puede mandar el mensaje a las instancias) desde instancias de otras clases. Como ya dijimos anteriormente, el método no devuelve ninguna respuesta, de ahí que esté marcado como **void**. Al igual que en la clase, el cuerpo del método esta demarcado con las llaves.

```
assertEquals(1+2, 3);
```

Finalmente tenemos la aserción. En ella encontramos la invocación al método estático (perteneciente a la clase Assert, recordemos el **import static**) assertEquals. Este método recibe dos parámetros que compara para saber si son iguales o no. En este caso recibe el resultado de evaluar 1+2 (en Java esto no es un envío de mensaje, sino que lo ejecuta directamente la máquina virtual) y el número 3. En Java, así como en muchos otros lenguajes, se evalúan primero los parámetros y luego con los resultados se envía el mensaje.



RESUMEN

A través de este capítulo pudimos conocer la historia de Java así como también algunas circunstancias que rodearon su aparición. Aprendimos los alcances de este lenguaje y vislumbramos el potencial que alcanza el tope de las grandes empresas desarrolladoras de software. Para continuar analizamos las características del entorno de desarrollo con la integración de Eclipse y lo utilizamos para crear nuestra primera aplicación, de esta forma, nos acercamos a la programación con Java.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Qué es una máquina virtual?
2	¿Por qué triunfó Java frente a otras tecnologías?
3	¿Cuál es la diferencia entre el JRE y SDK?
4	¿Qué es y para qué sirve Eclipse?
5	¿Qué es TDD?
6	¿Cuál es el principal beneficio de desarrollar utilizando TDD?
7	¿Que se utiliza JUnit?
8	¿Cómo se define un test case en JUnit?
9	¿De qué manera se define un test en JUnit?
10	¿En qué clase están definidas las aserciones?

ACTIVIDADES PRÁCTICAS

1	Investigue qué errores nos muestra Eclipse (en el panel de Problems, al salvar el código fuente) cuando borra distintas partes del código.
2	Trate de utilizar el compilador del SDK para el test que realizamos.
3	Pruebe ejecutar el resultado de la compilación con el comando java. Le dará error.
4	Cambie el test case que ya tiene, agregando nuevos métodos de test con diferentes aserciones. En el código fuente, dentro del método presione **CTRL + BARRA ESPACIADORA** para hacer aparecer el autocomplete y así ver las distintas clase de aserciones que hay.
5	Haga doble clic sobre el costado gris izquierdo del método, aparecerá un circulo azul, esto indica un breakpoint. Ejecute el test case en modo debug, eligiendo **Debug...** en vez de **Run...** Vea qué sucede.

*03 –

Sintaxis

La sintaxis es el conjunto de reglas que dan como resultado un programa bien escrito. Estas reglas describen qué elementos están permitidos en determinados contextos y cuál es la relación entre ellos. En este capítulo nos encargaremos de conocer y analizar la sintaxis básica que necesitamos para dominar la programación en Java y de esta forma comenzar a dar los primeros pasos en este interesante mundo.

- **Palabras claves..... 46**
- **Ciclos..... 52**
- **Declaraciones, expresiones, sentencias y bloques..... 58**
  - Variables..... 58
  - Sentencias..... 60
- **Otras estructuras..... 60**
- **Tipos primitivos y literales..... 67**
- **Operadores..... 70**
- **Paquetes..... 72**
- **Resumen..... 73**
- **Actividades..... 74**

Palabras claves

Las palabras claves o reservadas son palabras que Java no permite que los programadores utilicen libremente en cualquier parte del código (por ejemplo, los nombres de variables), sino que tienen un propósito determinado y un contexto específico. Eclipse nos indicará que una palabra está reservada porque estará resaltada con un color distinto del texto normal. A continuación encontraremos el listado de palabras claves, su explicación y uso:

- **abstract**: se utiliza en clases y métodos para indicar que son abstractos. Debemos tener en cuenta que en las clases esto significa que sirven para formar una jerarquía y no pueden tener instancias. En los métodos sirve para indicar que la jerarquía que tiene responde al mensaje, pero no define código alguno.
- **assert**: se utiliza para enunciar condiciones que se deben cumplir durante la ejecución. No confundir con las aserciones de JUnit.
- **boolean**: se encarga de representar al tipo primitivo booleano. Las variables de este tipo no son objetos y los únicos valores que pueden tomar son **true** (verdadero) y **false** (falso).
- **break**: es una etiqueta que se utiliza para salir de la ejecución de los ciclos (**for**, **while**, **do while**) y de las estructuras **switch** (ambas las veremos en detalle más adelante).
- **byte**: representa el tipo primitivo de números de 8 bits, entre -128 y 127. Lo valores de este tipo no son objetos.
- **case**: etiqueta usada en las estructuras **switch** para indicar un caso.
- **catch**: se utiliza en conjunto con la etiqueta **try** para indicar que se espera una excepción (las veremos más adelante) en el código envuelto en la estructura del **try**.



VALORES POR DEFECTO

Es interesante recordar que toda variable en el lenguaje de programación Java tiene un valor por defecto al momento de su creación, de esta forma nunca obtendremos una variable vacía al momento de crearla. En este sentido, los tipos primitivos numéricos y carácter se inicializan con el valor 0 mientras que los datos que corresponden al tipo denominado **boolean** se inicializan con el valor **false**. Por otra parte, cualquier referencia a objeto se inicializa con el valor **null**.

- **char**: se encarga de representar el tipo primitivo de los caracteres. Las variables que corresponden a este tipo no son objetos y de esta forma pueden contener cualquier carácter **Unicode**.
- **class**: define el comienzo de la declaración de una clase.
- **continue**: al igual que el **break**, es un etiqueta que se utiliza en los ciclos y sirve para hacer que se interrumpa la ejecución del código de la iteración actual del ciclo y que continúe con la siguiente.
- **default**: puede ser utilizada para definir una la acción por defecto en las estructuras **switch**.
- **do**: debemos tener en cuenta que se utiliza en combinación con el **while** para formar la estructura de ciclo **do while**.
- **double**: representa el tipo primitivo de los números de punto flotante de doble precisión (64 bits) definidos en el estándar **IEEE 754**.
- **else**: se encarga de representar el camino alternativo en una toma de decisión, se utiliza en conjunto con el llamado **if**.
- **enum**: nos permite declarar un tipo enumerado, clases que tienen un número fijo y definido de instancias nombradas.
- **extends**: se utiliza para indicar que una clase hereda de otra. Si no se utiliza, por defecto toda clase hereda de la clase **Object**. También se utiliza para declarar que una interfaz extiende otra (herencia entre interfaces).
- **final**: es un modificador que puede aplicar a clases, métodos variables y argumentos, y puede tener un significado distinto en cada caso. En las clases significa que se puede extender heredando. En caso de los métodos, que las subclases de la clase que lo contiene no pueden redefinirlo. En las variables (tanto las de instancia como las de clase, así como en las definidas en el cuerpo de los métodos) significa que se puede asignar valor sólo una vez. Finalmente, en el caso de los argumentos de los métodos, quiere decir que esa variable no se puede escribir (reasignarle un valor nuevo).
- **finally**: recordemos que se se usa en conjunto con la estructura **try**, y se encarga de especificar un bloque de código que se ejecuta, sí o sí, al finalizar la ejecución del bloque **try**.
- **float**: representa al tipo primitivo de los números de punto flotante de simple precisión (32 bits) definidos en el estándar **IEEE 754**.

> LAS PALABRAS RESERVADAS NO SE PUEDEN UTILIZAR LIBREMENTE

- **for**: es la etiqueta que define a la estructura de ciclo más usada.
- **if**: especifica una bifurcación en la ejecución del código guiada por una condición. Si la condición en verdadera (**true**) entonces se ejecuta el cuerpo del **if**, si no se lo saltea.
- **implements**: se trata de la manera de indicar que una clase implementa (hereda) de una o varias interfaces. La clase está obligada a implementar los métodos definidos por las interfaces que utiliza (a menos que sea una clase abstracta, en cuyo caso las encargadas serán las subclases que corresponden).
- **import**: declara el uso (importación) de clases y paquetes en un código fuente. Mediante el agregado del modificador **static**, también se pueden importar métodos **estáticos** de una clase.
- **instanceof**: instrucción que sirve para preguntar si un objeto es instancia de una determinada clase (o superclase) o interfaz. El resultado que se presenta es el valor **true** o **false**.
- **int**: representa el tipo primitivo de números de 32 bits, entre
- -2147483648 y 2147483647. Lo valores de este tipo no son objetos.
- **interface**: especifica el comienzo de la definición de una interfaz.
- **long**: se encarga de representar el tipo primitivo de números de 64 bits, se encuentran entre -9223372036854775808 y 9223372036854775807. Lo valores de este tipo no son objetos.
- **native**: modificador de método que indica que este no está implementado en Java sino en otro lenguaje, generalmente en **C++.** Se utiliza para hacer que Java se comunique con el exterior de la máquina virtual (por ejemplo, para acceder al sistema de archivos o a la red).
- **new**: es el operador de creación de instancias. Se utiliza junto con el nombre de la clase no abstracta que se quiere incluir en la creación de la instancia y los parámetros necesarios para la inicialización de esta. Como resultado se obtiene una nueva instancia de la clase.



SHORT CIRCUIT LOGIC

Tengamos en cuenta que esta técnica es muy utilizada para cuando queremos acceder a enviar mensajes a un objeto en una condición, pero no sabemos si la referencia esta apuntando correctamente (o sea, no utilizamos el valor **null** en esta operación). Recordemos que la estructura que corresponde es la que mencionamos a continuación: **if(referencia != null && referencia.hayQueSeguir())**.

- **package**: instrucción que define a qué paquete pertenece la clase. En Java la estructura de paquetes tiene que coincidir con la estructura de directorios.
- **private**: modificador que aplica a clases, métodos y atributos (de instancia y de clase). Indica cuál es el nivel de acceso al elemento. Aplicado a una clase (solamente para clases definidas dentro de clases) indica que solo se la puede utilizar dentro de la clase que la definió. En métodos, indica que solo se los puede invocar desde otros métodos de la clase que los define. En los atributos, significa que no pueden ser accedidos, salvo por los métodos definidos en la clase que los define. Tanto en el caso de los métodos como en el de los atributos, no pueden ser accedidos por otras clases, ni siquiera por subclases de la clase que los contiene.

> DEBEMOS USAR EL NIVEL DE VISIBILIDAD MÁS RESTRICTIVO, EVITANDO PONER TODO PÚBLICO

- **protected**: al igual que **private**, es un modificador de acceso y aplica a los mismos elementos, con la diferencia que este permite que los elementos sean también accedidos por subclases.
- **public**: es un modificador de acceso que indica que el elemento en cuestión es público y puede ser accedido por cualquier código. Aplica a clases, métodos y atributos.
- **return**: es la instrucción que indica que se termina la ejecución de un método y, opcionalmente, devuelve un valor como respuesta.
- **short**: representa el tipo primitivo de números de 16 bits, entre -32768 y 32767. Lo valores de este tipo no son objetos.
- **static**: tiene tres usos bien distintos. El primero y más importante es indicar que un método o un atributo pertenecen a la clase en vez de ser de instancia, se dice que son estáticos. El segundo, es especificar la importación de los métodos estáticos de una clase usándolo en conjunto con **import**. Finalmente, el tercer y último uso es al declarar clases e interfaces dentro de otra clase (o interfaz). La diferencia entre indicar que sea estática o no la clase (o interfaz) interna es que puede ser accedida como cualquier elemento estático, sólo utilizando el nombre de la clase contenedora. De otra forma se requiere de una instancia para tener acceso a la clase.
- **strictfp**: antes de la versión 1.2 de la **JVM** todas las operaciones con punto flotante daban como resultado valores estrictos respecto

del estándar **IEEE 754**. Pero a partir de esa versión, Java utiliza para los cálculos intermedios otras representaciones que puedan estar disponibles en la máquina. De esta manera se obtiene mayor precisión en las operaciones, pero los resultados no son transportables. Si deseamos forzarlos a que sí lo sean, se puede utilizar este modificador de método.

- **super**: es un identificador que está relacionado con **this** en que también es una referencia al objeto actual (receptor del mensaje que se está ejecutando). Pero la diferencia radica en que se utiliza para indicar que se quiere ejecutar un método que se encuentra en alguna de las superclases de la clase del método que se está ejecutando. Es una marca para modificar el comportamiento de **method lookup** para que no comience la búsqueda del método en la clase propia del objeto receptor. También se utiliza para especificar un cota inferior a los tipos de un genérico (los veremos más adelante).
- **switch**: sirve para declarar una estructura **switch**. Dicha estructura se utiliza para comparar una variable de tipo primitivo con ciertos valores y, en caso de que coincida con alguno, ejecutar cierto código. Se utiliza con las etiquetas **case**, **break** y **default**.
- **synchronized**: es tanto un modificador como una instrucción. Como modificador, aplica sólo a los métodos de instancia y tiene como objetivo sincronizar el acceso concurrente de distintos códigos ejecutándose en paralelo. Lo que hace es forzar que sólo un código (**threads** o hilos de ejecución, son procesos livianos) pueda estar ejecutando métodos sincronizados del objeto. Los threads son frenados y esperan a que el thread en ejecución termine y así le toca al siguiente. La forma de instrucción sirve para esencialmente lo mismo, pero se utiliza dentro del cuerpo de los métodos y delimita la zona de código que quiere sincronizar. Además requiere que se especifique cuál es el objeto que se va a utilizar para realizar la



ESTRUCTURAS VERSUS MENSAJES

Las estructuras como el **if** o el **for** están en la sintaxis del lenguaje y no representan ningún envío de mensajes. Esto hace que sean un elemento extraño al paradigma. Algunos lenguajes, como **Smalltalk**, tienen el mismo comportamiento pero al enviar mensajes. Lo que los hace más flexibles.

sincronización (debemos tener en cuenta que en el caso de ser un modificador, el objeto utilizado era el receptor del mensaje).

- **this**: es una referencia a la instancia actual que está ejecutando el método en cuestión. Es la instancia receptora del mensaje que corresponde con la ejecución del método. Solamente se puede utilizar en el cuerpo de los métodos de instancia.
- **throw**: se trata de la instrucción que podemos utilizar para lanzar una excepción durante la ejecución.
- **throws**: indicador de que un método lanza excepciones. El compilador se asegura de que el emisor del mensaje que origina una excepción de estas la atrape con la instrucción **catch** o que indique que también la lanza (deja pasar) al anotarse en conjunto con **throws** y el nombre del la clase de la excepción.
- **transient**: es un modificador de atributos que los marca como no serializables. Significa que cuando un objeto es convertido a bytes para ser transmitido por red o a un archivo, los atributos marcados como transient son ignorados. En el proceso de reconstrucción, estos son inicializados a su valor por defecto.
- **try**: etiqueta utilizada para delimitar un bloque de código. Se utiliza en conjunto con las etiquetas **catch** y/o **finally**.
- **void**: indica que un método no devuelve nada como respuesta.
- **volatile**: es un modificador de atributos (tanto de instancia como de clase) que indica que el atributo es accedido y modificado por varios threads. Si bien todo atributo puede ser accedido concurrentemente, la JVM toma recaudos con estos atributos forzando que las operaciones de escritura y de lectura de estos sean atómicas.
- **while**: etiqueta usada para los ciclos **while** y **do while**.

> MUCHAS PALABRAS CLAVES NO SE USAN CASI NUNCA, SALVO EN CASOS MUY PARTICULARES

Es importante saber que Java posee un par de palabras reservadas que si bien no se utilizan, se encuentran en la definición del lenguaje. Ellas son **const** y **goto**, ambas son resabios de versiones primitivas del lenguaje. Así mismo se definen también las siguientes palabras claves que son objetos conocidos, se los llama literales.

- **false**: debemos tener en cuenta que se trata del elemento que representa la opción de falsedad, su tipo corresponde a **boolean**.

- **true**: es el elemento que representa la verdad, su tipo es **boolean**.
- **null**: se encarga de representar el vacío, es decir, la nada. Solo las referencias a objetos pueden estar asignadas a **null** (los valores primitivos no). **null** no tiene tipo (no es instancia de ninguna clase) pero puede ser asignado a cualquier referencia o pasado como parámetro cuando se espera un objeto.



Ciclos

Los ciclos son estructuras de código que nos permiten representar situaciones como "mandar a cada cliente un e-mail con las ofertas del mes" o "mientras haya papas en el cajón, tengo que pelarlas". En Java tenemos cuatro estructuras de ciclos, todas con el mismo poder, pero cada una con una expresividad distinta. Tenemos el ciclo **for**, el **while**, el **do while** y el **for each**.

El ciclo for

El ciclo for (para) es una de las estructuras más usadas en los lenguajes de programación. Consta de cuatro partes: la inicialización, la condición, el paso siguiente y el cuerpo del ciclo. Veamos cómo se ve esto en Java.

```
int sum = 0;
for(int i = 0; i < 4; i++) {
    sum += i;
}
```

El fragmento de código anterior suma los números 0, 1, 2 y 3 en la variable **sum**. La primera línea es la declaración e inicialización de la variable **sum** en 0. Luego tenemos el ciclo **for** donde podemos apreciar la inicialización (**int i = 0**) dónde declaramos otra variable. Esta variable solo puede ser accedida por el código **for**. Separada por un punto y coma tenemos la condición (**i < 4**), que dice si la variable **i** es menor a 4. Después, también separado por un punto y coma, encontramos el paso siguiente (**i++**) que en este caso dice "incrementar en 1 la variable **i**".

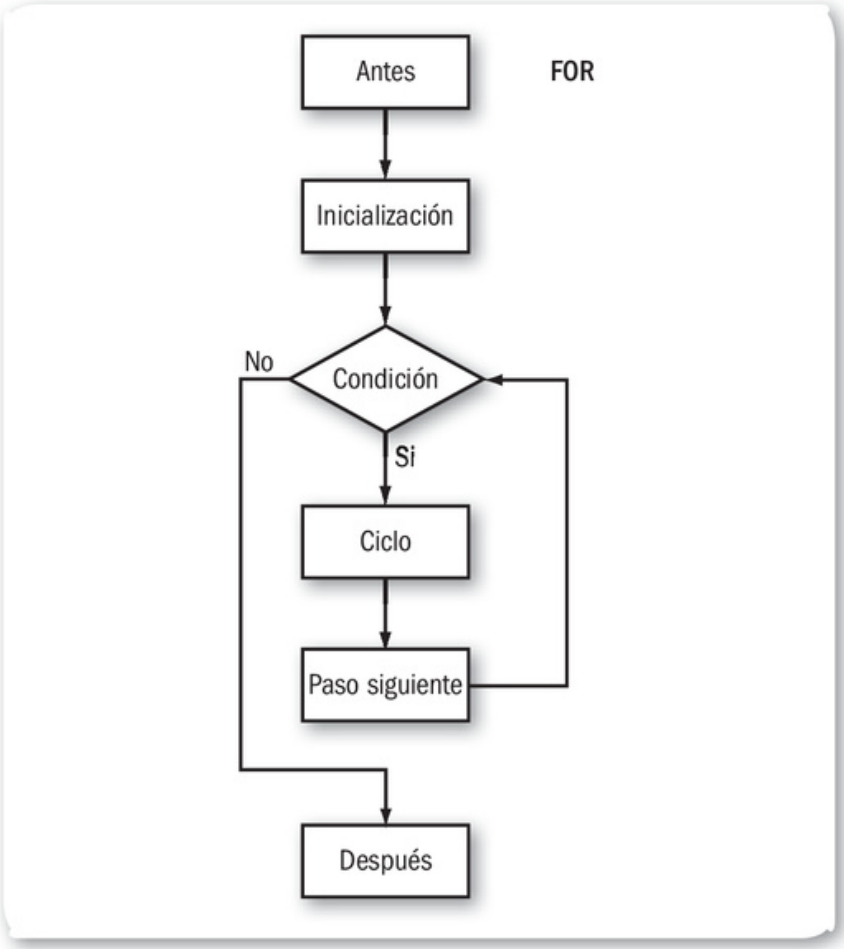


Figura 1. En esta imagen podemos ver el diagrama de flujo que representa el funcionamiento del ciclo **for**.

El cuerpo del **for** es el código comprendido entre las llaves. En él encontramos la sentencia **sum += i;** que es igual a escribir **sum = sum + i;**, o sea, sumar lo que hay en **sum** con lo que hay en **i** y el resultado guardarlo en **sum**. La semántica de **for** es la siguiente: "para **i** igual a 0, mientras **i** sea menor que 4, sumar **sum** e **i** y guardarlo en **sum**, luego incrementar **i** en 1 y repetir". Formalmente, la estructura del **for** acepta cualquier sentencia válida de Java como inicialización (incluso nada) y se ejecuta solamente una vez al inicio del **for**. Acepta cualquier sentencia que dé como resultado un valor **boolean** como condición (nada significa **true**) que se pregunta al inicio de cada iteración (paso o ciclo) y ejecuta el código si el resultado es **true** e interrumpe el **for** si es **false**. Finalmente cualquier expresión valida para el paso siguiente,

que se ejecuta al final de cada ejecución del código del cuerpo. Dentro del cuerpo del ciclo es posible utilizar las etiquetas **break** y **continue** para alterar el comportamiento de este. La etiqueta **break** se usa para abortar la ejecución del **for** completamente, en cambio, la etiqueta **continue** se utiliza para terminar la ejecución de la iteración actual y ejecutar la siguiente. Veamos un ejemplo:

```
int sum = 0;
for(int i = 0; i < 4; i++) {
    if(i == 1) continue; // si i es igual a 1 ir al paso siguiente
    if(i == 3) break; // si i es igual a 3 abortar el for
    sum += i;
}
assertEquals(sum, 2);
```

El ciclo while

Debemos tener en cuenta que el ciclo while es más simple que **for** dado que solamente tiene la condición y el cuerpo. De esta forma, un ciclo **while** que sume los números del 0 al 3 se ve así:

```
int sum = 0;
int i = 0;
while(i < 4) {
    sum += i;
    i++;
}
```

Como vemos en el bloque de código anterior, si nos encargamos de realizar la comparación con el correspondiente al del ciclo **for**, tenemos la inicialización fuera de la estructura de **while** y el incremento de la variable **i** se encuentra dentro del cuerpo. A **while** hay que leerlo de esta forma: **mientras se cumpla la condición, ejecutar el cuerpo**. Como en el **for**, y en todas las otras estructuras de ciclos, podemos utilizar las etiquetas **break** y **continue**.

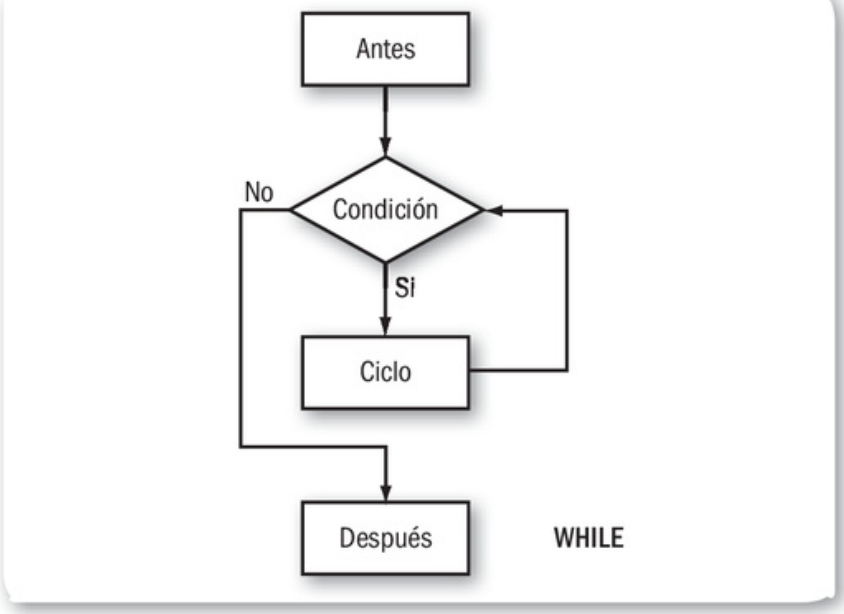


Figura 2. En esta imagen podemos ver el diagrama de flujo correspondiente al funcionamiento del ciclo **`while`**.

El ciclo do while

El ciclo do while es una variación del ciclo **while**, donde primero se ejecuta el cuerpo y posteriormente se pregunta si se cumple la condición para continuar iterando o no, veamos un ejemplo:

```
int sum = 0;
int i = 0;
do {
sum += i;
i++;
} while(i < 4);
```

Hay que ser cuidadosos a la hora de usar esta estructura, porque el cuerpo se ejecuta sí o sí al menos una vez. Por esta razón debemos ser muy cuidadosos si estamos trabajando en u proyecto agregamos un ciclo **while**,y posteriormente decidimos transformarlo en un **do while** ya que podemos tener dificultades,

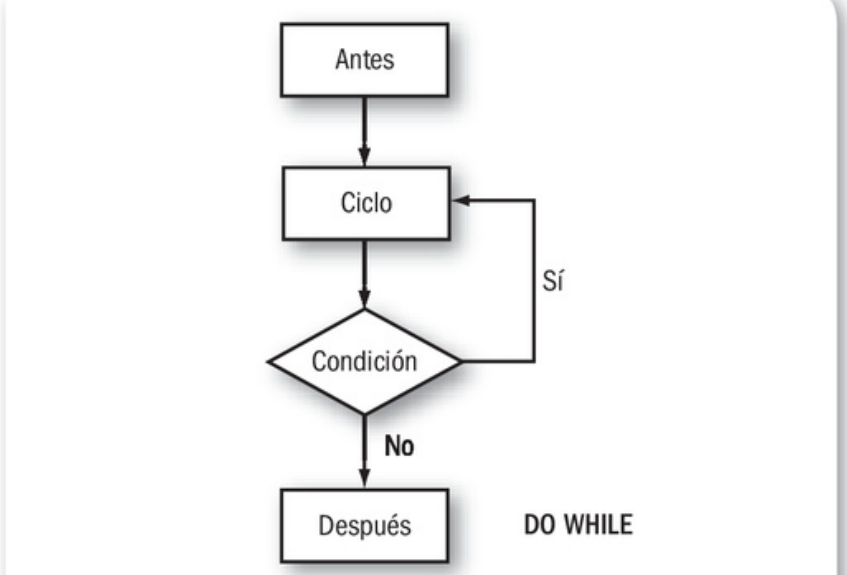


Figura 3. El diagrama de flujo que vemos en esta imagen corresponde al ciclo conocido como **do while**.

El ciclo for each

Este ciclo sirve para recorrer estructuras tales como listas y otras colecciones de objetos. Estas estructuras se conocen como iterables y responden al mensaje **iterator** que devuelve una instancia de **Iterator**, que es un objeto que sabe cómo recorrer la estructura. El for each es una variación del **for** para ser usada con estas estructuras iterables:

```
// asumamos que dígitos es una colección con los números del 0 al 9
...
int sum = 0;
for(int digito : digitos) {
}
assertEquals(sum, 45);
sum += digito;
```

Leemos este código de esta forma: por cada **digito** en **digitos**, sumar **sum** con **digito** y guardarlo en **sum**.

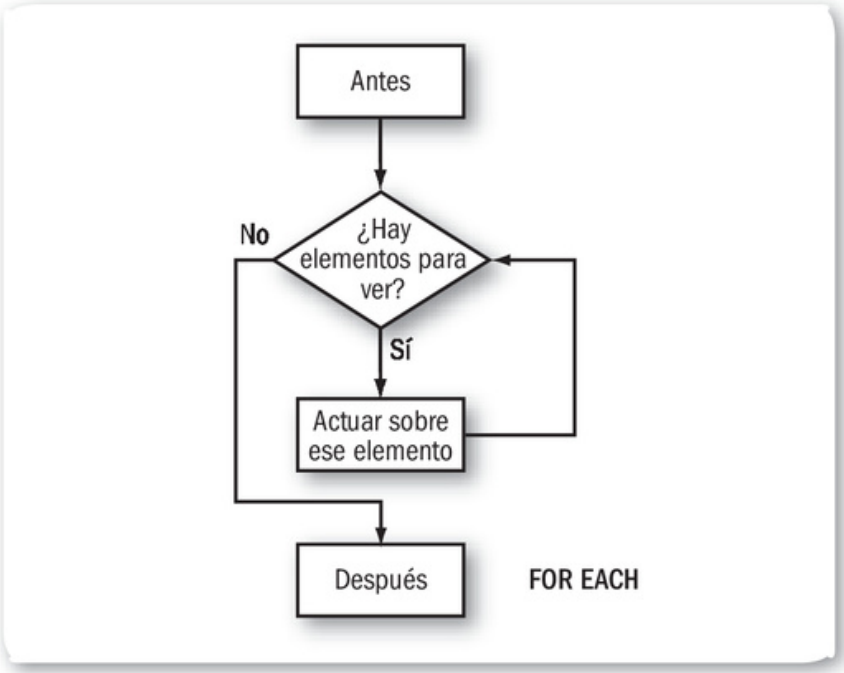


Figura 4. En esta imagen vemos el diagrama de flujo correspondiente al ciclo **for each**.

Realmente esta estructura es lo que se conoce como **syntax sugar** (endulzar la sintaxis, dado que no agrega funcionalidad, solamente facilidad en el uso de algo que ya estaba) del **for** común, como vemos en el siguiente fragmento de código.

```
int sum = 0;
for(Iterator i = digitos.iterator; i.hasNext(); ) {
    int digito = ((Integer) i.next()).intValue();
    sum += digito;
}
assertEquals(sum, 45);
```

Como podemos observar, el programador tiene que escribir más código (y por lo tanto, con mayor posibilidad de cometer errores) y es menos legible que la versión del **for each**. Esta es la traducción del **for each** al **for** que realiza el **compilador** Java y que hasta la versión 1.5 los programadores tenían que escribir. En la inicialización

se crea una variable del tipo **Iterator** y se la asigna con el iterador de la colección de los dígitos. La condición pregunta si existe un próximo elemento que visitar, si existe otro dígito. El paso siguiente se encuentra vacío dado que lo realizaremos en el cuerpo con el envío del mensaje **next** al iterador. En el cuerpo conseguimos el siguiente elemento con **next** y se lo asignamos a la variable dígito (la que creamos en cada paso del **for each**) y realizamos la suma que corresponde. Notar que forzamos el resultado de **next** a **int** con la instrucción **Integer** y el mensaje **intValue**. Lo primero se conoce como **casteo** donde le indicamos a la **JVM** que sabemos el tipo real del objeto y queremos usarlo (**next** devuelve un **Object** y no se puede asignar directamente a **int**). Lo segundo Java lo hace automáticamente en el caso del **for each** y se conoce como **auto boxing** (envolver y desenvolver un dato primitivo en su clase asociada).



Declaraciones, expresiones, sentencias y bloques

Veremos a continuación las distintas piezas básicas del lenguaje para que podamos realizar la construcción del código deseado.

Variables

En Java se conoce como variable a todo colaborador interno (variables de instancia y de clase), externos (parámetros) y a los nombres locales en los métodos (variables locales). Existen algunas reglas para nombrar a las variables:

- El primer carácter debe ser una letra, el guión bajo (_) o el signo de dólar (**$**, aunque está reservado).
- Puede estar seguido de letras, números, el guión bajo o el signo dólar.
- No hay límite para la longitud del nombre.
- No debe ser una palabra clave.

Lo más conveniente es que los nombres de las variables reflejen el significado del dato que contienen (usar **velocidad** en vez de **v**). Si el

nombre es una serie de palabras, la primera letra de las palabras van en mayúscula (**diaDelMes**), esto es una convención. Las constantes se escriben en mayúsculas separando las palabras por guiones bajos.

Una variable, se define especificando primero su tipo y luego su nombre. Así mismo se le pueden aplicar modificadores como **final**, que hace que la referencia o el valor (se indica antes del tipo) solamente se pueda asignar una vez.

> LAS CONSTANTES SE ESCRIBEN EN MAYÚSCULAS SEPARADAS POR GUIONES BAJOS

Expresiones

Una expresión es una combinación de variables, operadores y envíos de mensajes que evalúan un valor. El tipo del valor dependerá de los tipos de los elementos involucrados. Veamos algunos ejemplos.

```
speed = auto.getSpeed()
```

Aquí tenemos varias expresiones, primero tenemos auto, una expresión que devuelve el objeto referenciado (supongamos tipo **Car**); luego el envío del mensaje **getSpeed** a auto, que devuelve un **double**; finalmente, la asignación es otra expresión que devuelve el valor que se le asigna a **speed** y el tipo es del tipo de la variable. Más ejemplos.

```
1 + 2 * 3
"hola" + ' ' + "mundo"
true != false
esPar(3) ? "par" : "impar"
```



BLOQUES Y SENTENCIAS

Tengamos en cuenta que los ciclos y las estructuras como el **if** que requieren un bloque de código para trabajar, pueden recibir una única sentencia en vez de un bloque. En estos casos es posible obviar las llaves. Aunque por un tema de claridad es mejor incluirlas siempre.

Sentencias

Las sentencias son la unidad completa de ejecución. En general finalizan con un punto y coma (;) que las separa, salvo en el caso de los ciclos y las estructuras.

```
speed = auto.getSpeed();
Fruta naranja = new Naranja();
unValor++;
auto.stop();
```

Bloques

Un bloque es una secuencia de sentencias encerradas entre llaves ({ y }) y puede ser usado en cualquier lugar donde va una sentencia (ya que un bloque es un sentencia). Los bloques en general se utilizan como el cuerpo de las estructuras (como los ciclos) aunque también se los puede utilizar solos, ya que definen un alcance léxico para los nombres. Esto quiere decir que se puede redefinir una variable en un bloque, así el código del bloque accede a esta y el código fuera del bloque accede a la primera definición.



Otras estructuras

Las estructuras son construcciones puramente sintácticas que se traducen en funcionalidad. Un ejemplo de estructura son los ciclos que ya vimos. Ahora veamos otras estructuras que nos ofrece Java.

if/else/else if

Esta es una de las estructuras de control de flujo más básicas de los lenguajes de programación. Esta estructura permite, en base a una condición, modificar el flujo de ejecución y se utiliza para situaciones como “si llueve no salgo, si no, salgo”.

Representa una bifurcación, la toma de decisión donde solo hay dos opciones, si sí o si no. A continuación veamos cómo se escribe:

```
...
if(estaLloviendo()) {// si da false continúa en A
        noSalgo();
...
// al finalizar continúa en A
}
...
// A
```

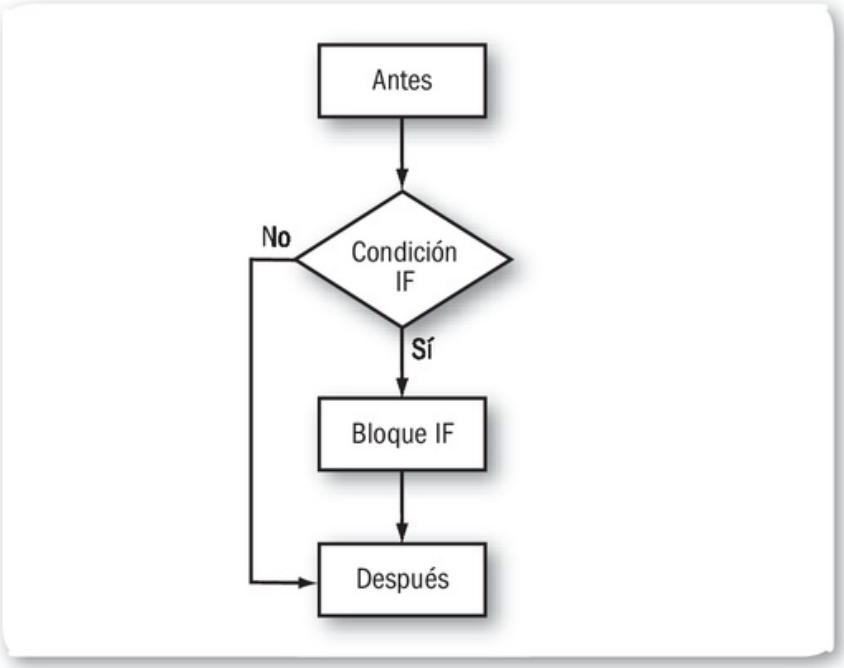


Figura 5. En esta imagen podemos ver el diagrama de flujo correspondiente al funcionamiento del ciclo `If`.

En el ciclo **if** se evalúa la condición booleana que se encuentra entre los paréntesis y en base al resultado se ejecutan las acciones que se encuentran en el cuerpo de **if** (en caso de **true**). Si no, continúa con la ejecución ignorando el **if**. Si queremos que se ejecute un código en el caso de **false** (manteniendo el caso de **true**), utilizamos la etiqueta **else** después del cuerpo de **if**, seguido de un bloque de código.

Siempre es mejor y más claro poner la opción más frecuente como bloque **if** ya que es lo primero otra persona va a leer.

```
if(estaLloviendo()) {// caso true
    noSalgo();
    ...
    // al finalizar continúa en A
} else { // caso false
    salgo();
    ...
    // al finalizar continúa en A
}
...
// A
```

Finalmente, podemos encadenar varios **if** utilizando **else if(<condicion>)**, de acuerdo a nuestras necesidades.

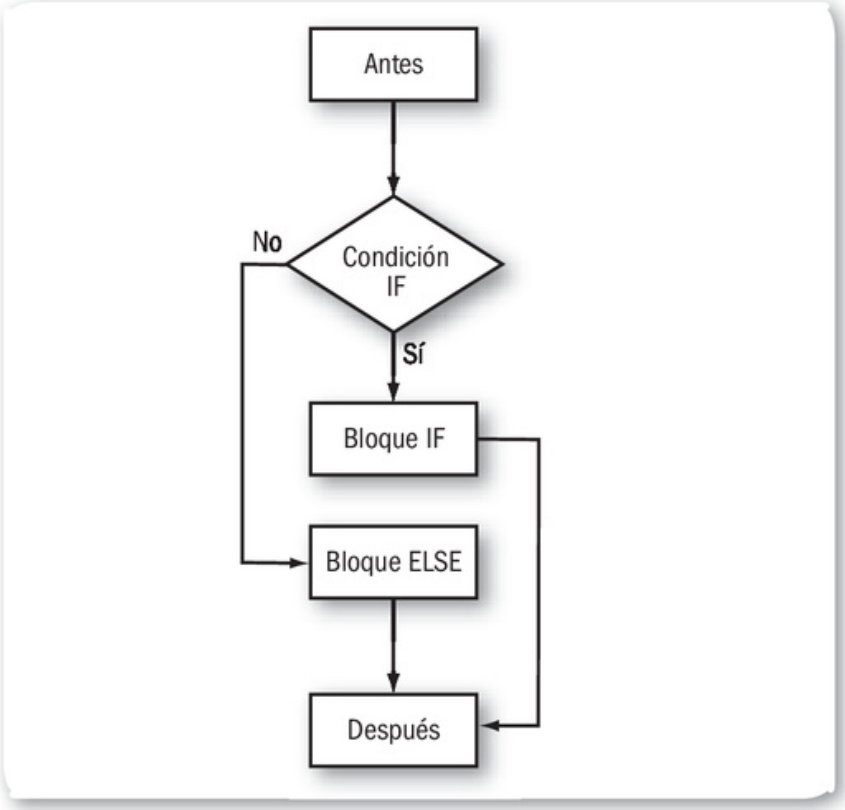


Figura 6. En esta imagen podemos ver el diagrama de flujo correspondiente al funcionamiento de `ifElse`.

Switch

Debemos tener en cuenta que esta estructura de control de flujo nos permite realizar la comparación de un tipo numérico entero y carácter primitivo contra varios candidatos que deseemos y que de esta forma podremos ejecutar el código en consecuencia.

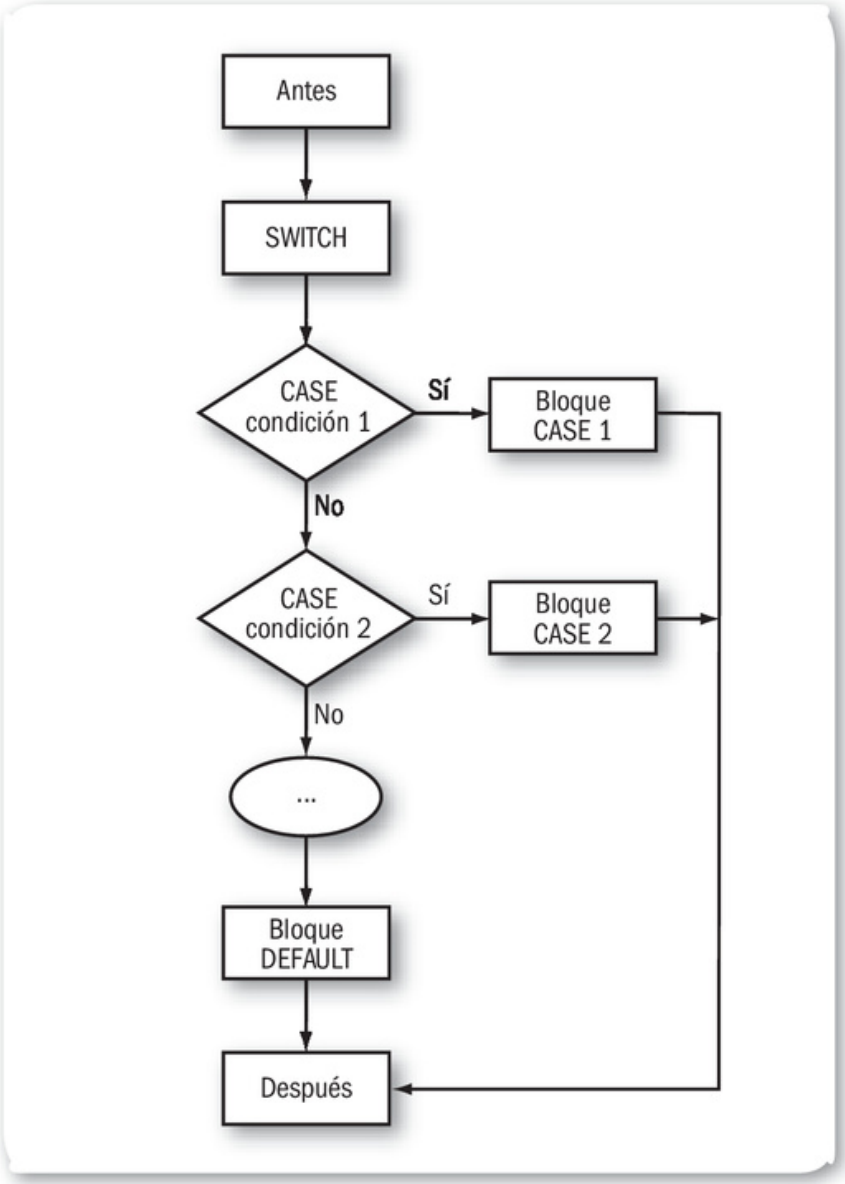


Figura 7. En esta imagen podemos ver el diagrama de flujo correspondiente al ciclo denominado **`switch`**.

La forma de escribirlo es presentada a continuación:

```
switch(<valor>) {
case <literal>: <código> break;
case <literal>: <código> break;
...
default: <código> break;
}
```

Al switch le pasamos un valor para comparar, luego, en cada caso, lo comparamos contra un literal del mismo tipo y, en caso de ser iguales, se ejecuta el código asociado. Se pone la etiqueta **break** al final del código de cada caso para poder finalizar la ejecución del **switch** (generalmente es lo deseado), si no se seguirían ejecutando en cascada el resto de los códigos del **switch** (generalmente es un bug). El caso **default** se ejecuta cuando ninguno de los casos coincidió.

```
int opcion = 2;
switch(opcion) {
case 1: ...
// código para el caso 1
// sin break, continúa ejecutando el caso 2
case 2: ...
// código para el caso 2
break;
// continúa en A
default: ...
// código para cuando no es ni 1 ni 2
break;
// continúa en A
```



ESPACIOS EN BLANCO

Debemos tener en cuenta que el lenguaje Java utiliza cualquier espacio blanco (espacios, saltos de líneas y sangrías; uno o más de los anteriores) para separar las palabras del código. Son necesarios a menos que haya algún otro símbolo que separe (paréntesis, comas, llaves, etcétera).

```
}
...
// A
```

Try/catch/finally

Esta estructura está generalmente asociada al manejo de excepciones, aunque puede ser usada para otros fines. El objetivo es que se ejecuta el código del bloque seguido a try y pueden pasar dos cosas, que el código se ejecute exitosamente o que ocurra una excepción.

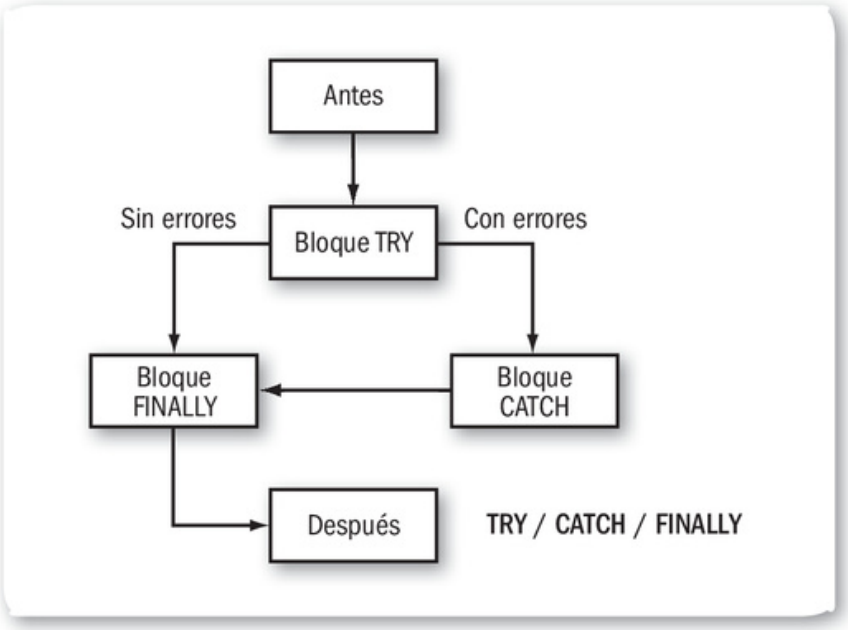


Figura 8. Diagrama de flujo que representa el funcionamiento del ciclo **`try catch finally`**.

Debemos tener en cuenta que si nos encragamos de especificar que se realice la acción de atrapar una excepción con la etiqueta catch, entonces se procederá a ejecuta el código asociado a esta. Si finalizó la ejecución en forma correcta o se terminó de ejecutar el código que corresponde a **catch**, y tuvimos cuidado de especificar código con la etiqueta **finally**, se evalúa lo que corresponde. Veremos esta estructura en más detalle más adelante, en la sección de excepciones.

```
try {
    …
    throw new Exception("ocurrio un error"); // continúa en B
    …
    // si finaliza bien continúa en A
} catch(Exception e) {
    …
    // B, continúa en A
}
…
// A
…
try {
    …
    throw new Exception("ocurrio un error"); // continúa en B
    …
    // si finaliza bien continúa en B
} finally {
    …
    // B, continúa en A si término bien
}
…
// A
…
try {
    throw new Exception("ocurrio un error"); // continúa en B
    …
    // si finaliza bien continúa en C
} catch(Exception e) {
    // B, continúa en C
} finally {
    …
    // C, continúa en A si término bien
}
…
// A
```

Synchronize

Es una estructura de sincronización de threads. Se usa para asegurarse de que un solo thread está ejecutando el código del bloque asociado. Para ello se utiliza un objeto a modo de llave, solamente un único thread puede tener la llave en un momento dado. El thread que tiene la llave es el que puede ejecutar. El resto espera hasta que se libere la llave.

```
Object key = new Object();
...
// este código puede ser ejecutado en paralelo por varios threads
synchonize(key) {
...
// este no
}
```

Tipos primitivos y literales

Si bien Java es un lenguaje orientado a objetos, posee valores que no son objetos. Estos elementos son los llamados tipos primitivos. Los tipos primitivos son los representantes de los números y los caracteres que debido a problemas en la performance de las primeras versiones de Java, se decidió que no sean objetos. Son valores bien conocidos por la máquina virtual y cuyas operaciones están cableadas en ella. Esto presenta un desafío para el programador dado que tiene que lidiar con objetos y con los tipos primitivos. Al no ser objetos (no ser referenciados) no se les puede asignar null, ni utilizar en lugar de cualquier objeto. No son instancia de ninguna clase, no tienen métodos ni responden a mensajes. Solo se pueden realizar operaciones sobre ellos como la suma y resta, y otras que están definidas por el lenguaje directamente. Para tratar de salvar un poco estos problemas, Java introduce clases que envuelven a los tipos primitivos en objetos que,

> AUNQUE JAVA ES UN LENGUAJE ORIENTADO A OBJETOS, POSEE VALORES QUE NO SON OBJETOS

además, incluyen algunos métodos para transformar del tipo primitivo a texto y viceversa. Los tipos primitivos son:

- **boolean**: representa a los valores verdadero y falso de la lógica de **Bool**. Lo valores de este tipo son **true** y **false**. La clase asociada con este tipo primitivo es **Boolean**.

```
boolean verdadero = true;
```

- **byte**: se encarga de representar a los números de 8 bits entre -128 y 127. La clase asociada es la denominada **Byte**.

```
byte eñe = 164; // el ascii de la ñ
```

Recordemos que en el lenguaje de programación Java los números enteros se pueden escribir tanto en **decimal** (164) como en **octal** (0244) y en **hexadecimal** (0xA4 o 0xa4).

- **char**: es el tipo que representa a un carácter **Unicode** (16 bits). La clase asociada es **Character**. Los literales de carácter en Java se escriben entre comillas simples, y aceptan un carácter o el código Unicode. También aceptan un número.

```
char eñeCaracter = 'ñ';
char eñeNumero = 241; // código unicode decimal
char eñeUnicode = '\u00F1';
```

Para caracteres como el retorno de carro y la sangría, Java tiene unos códigos especiales que sirven también para las cadenas de texto, se les dice caracteres escapados.

```
char lineaNueva = '\n';
char comillaSimple = '\'';
char comillaDoble = '\"';
char lineaNueva = '\n';
char sangria = '\t';
char barraInvertida = '\\';
```

- **double**: es el tipo de los números de punto flotante de 64 bits de precisión definidos en el estándar **IEEE 754**, cuya clase asociada es **Double**. Los literales se pueden escribir con los decimales (usando el punto como separador) o en notación científica.

```
double unValor = 123.4;
double otroValor = 1.234e2; // mismo valor pero en notación científica
doublé yOtroMas = 123.4d; // especifico que es doublé el literal
```

- **float**: similar al **double** pero de 32 bits de precisión, su clase asociada es **Float**. Los literales se pueden escribir como los de **double** (sin la d) aunque ahí podríamos tener un problema al transformar un **double** literal a un **float**, lo mejor es especificar que es un literal de **float** usando la f.

```
float literal = 123.4f;
```

- **int**: se encarga de representar los números de 32 bits enteros, que se encuentran entre los valores -2147483648 y 2147483647. Su clase asociada es **Integer**. Los literales se pueden escribir de la misma forma que los literales de **byte**.
- **long**: representa los números de 64 bits, entre -9223372036854775808 y 9223372036854775807. Su clase asociada es **Long**. Sabemos que los literales se pueden escribir en decimal, octal o también en hexadecimal.
- **short**: representa los de números de 16 bits, entre -32768 y 32767. Su clase asociada es **Short**. Los literales se pueden escribir de la misma forma que los literales de **long**.
- **String**: si bien **String** no es un tipo primitivo, ya que es una clase, la incluimos en este apartado dado que el lenguaje da soporte para manejar las cadenas de caracteres de forma similar a un dato primitivo. En los literales aplican también las reglas de de escape vistas para los char y se crean utilizando la comillas dobles (“”).

```
String texto = "hola mundo";
```

- **Arrays**: los arrays (arreglos) son colecciones indexadas de elementos de tamaño fijo y establecido en el momento de creación. El acceso a los elementos de un array está dado por un índice que va desde 0 hasta la posición n-1, donde n es la longitud del array. Los arrays son instancias de la clase Array. Los arrays se pueden crear de la siguiente manera:

```
String [] unArray = new String[4]; // un array con 4 posiciones vacías
String [] otroArray = new String [] {"a", "b"} //
```

El acceso a los elementos de los arrays se realiza con el operador [] y un índice entero, podemos ver la forma correcta de realizar esta operación en el siguiente bloque de código:

```
assertEquals(otroArray[0], "a");
otroArray[0] = "c";
assertEquals(otroArray[0], "c");
otroArray[10] = "error"; // arroja una excepción de tipo
ArrayIndexOutOfBOundException
```



Operadores

Java permite operar sobre los tipos numéricos primitivos, usando los conocidos operadores matemáticos más algunos otros. Estos operadores realizan funciones que están cableadas en la máquina virtual de Java y no son envíos de mensajes. Los operadores son:

Operadores aritméticos

- **+, -, /, ***: suma, resta, división, multiplicación.
- **%**: módulo.
- **-**: negación de un número.

El operador **+** también sirve para concatenar varias **String**.

Operadores lógicos

- **&, &&, |, ||, ^, !**: y, y (**short circuit**), o inclusivo, o inclusivo (**short circuit**), o exclusivo, que corresponde a negación.
- **==, !=**: igualdad, desigualdad. Estos operadores evalúan la igualdad de tipos primitivos, **4 == 3 + 1** da como resultado **true**; o de referencias para los objetos, si una variable hace referencia al mismo objeto. No verifica que dos objetos sean iguales (que se puedan intercambiar), eso se verifica enviando el mensaje **equals** a un objeto mediante el envío del objeto para comparar.
- **<, <=, >, >=**: menor, menor o igual, mayor, mayor o igual. Sirven para comparar valores numéricos primitivos entre sí.

Los operadores de **short circuit** se utilizan cuando no queremos evaluar la segunda parte de la condición, cuando con el resultado de la primera parte es suficiente para saber el resultado general.

Operadores de bit

- **&, |, ^**: y, o inclusivo y o exclusivo. Operan sobre los bits que conforman los tipos primitivos.
- **~**: complemento bit. Invierte los bits de un valor.
- **<<, >>, >>>**: operadores de corrimiento de bits. Mueven los bits hacia la izquierda o derecha, tantos bits como se indiquen. En la práctica es igual a multiplicar o dividir por 2 (mantienen el signo, salvo **>>>**).

Otros Operadores

- ++, --: incrementar o disminuir en 1 una variable de tipo numérica entera (**byte**, **short**, **int** y **long**). Puede ser tanto sufijo como prefijo, en el primer caso, primero se modifica el valor y luego se lo lee; en el



CASTEOS

Tengamos en cuenta que los casteos en Java no son bien vistos ya que indican un error en el modelado. Hay casos particulares donde su uso es inevitable, uno de ellos es cuando se utilizan las colecciones de forma no genérica (código viejo), donde sólo se utiliza **Object**.

segundo, se lee y luego se modifica. Son operadores unarios.
- =: asignación, se utiliza para asignar un valor a una variable. Devuelve el valor asignado.
- ?:: operador ternario, recordemos que se utiliza como forma corta de la estructura que corresponde a if/else.
- instanceof: operador que chequea si un objeto es instancia de una determinada clase (se consideran también las superclases).



Paquetes

Los paquetes son la manera que tiene Java para organizar las clases en agrupaciones que tengan sentido. Los paquetes sirven para importar las clases que usamos. También para permitir qué clases, que representan cosas distintas pero se llaman igual, pueden coexistir (por ejemplo la clase Punto, para geometría y la clase Punto para gráficos 3D). Los paquetes en Java siguen la misma estructura de directorios que contiene a los archivos fuentes. Para crear un paquete, tan solo debemos crear una clase que indique que pertenece a ese paquete, y respetar la estructura de directorios. Para ello utilizamos la palabra clave **package** seguida del nombre completo del paquete (paquetes padres y su nombre al final). Debe ser la primer instrucción del archivo fuente.

```
package padre1.padre2.padre3.nombre;
```

En el ejemplo deberíamos tener la estructura de directorios correspondientes a **padre1**, dentro de e ste un directorio **padre2**, dentro, otro llamado **padre3** y, finalmente, dentro de este último el directorio **nombre** con el archivo fuente de la clase en él.

Para importar todas las clases de un paquete debemos declararlo utilizando **import** y el nombre completo del paquete, e indicando con un asterisco (*) que importamos todas la clases que están en el nivel indicado (no se importan las clases que estén en subpaquetes).

```
import red.user.java.*;
```

También podemos indicar que importamos solamente una clase.

```
import red.user.java.ClaseParticular;
```

Las instrucciones de **import** deben ir después de la declaración de paquete y antes de la definición de la clase. Finalmente podemos importar todos los métodos estáticos de una clase si agregamos el modificador **static** a la instrucción **import** de una clase.

```
import static red.user.java.ClaseConMetodosEstaticos;
```

Nosotros hemos utilizado este tipo de importación en los unit test.



RESUMEN

En este capítulo hemos conocido la sintaxis y la semántica del lenguaje Java. Estas son las reglas que permiten de nir programas que sean correctos desde el punto de vista del lenguaje (compilador y máquina virtual). Estas reglas por sí solas no hacen que un programa sea correcto en cuanto a la tarea que tiene que realizar. Somos nosotros (con ayuda de algunas herramientas como los unit test) los encargados de veri car que la lógica del programa sea correcta. Todas estas reglas las iremos aplicando en los sucesivos capítulos y las asimilaremos mediante su uso.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Para qué se usa final?
2	¿Qué diferencias hay entre el while y el do while?
3	¿Para qué sirve la estructura if/else/else if?
4	¿Qué son las expresiones?
5	¿Qué es una sentencia?
6	¿Un bloque es una sentencia?
7	¿En qué difiere un import static de uno normal?
8	¿De qué forma se relacionan el **for** normal y el **for each**?
9	¿Cuáles son las palabras claves relacionadas con el manejo de excepciones?
10	¿Con que tipo de dato se pueden utilizar los operadores lógicos?

ACTIVIDADES PRÁCTICAS

1	Crear un array, recorrerlo usando el ciclo for y operar sobre los elementos.
2	Crear un array, recorrerlo usando el ciclo for each y operar sobre los elementos.
3	Acceder a una posición invalida de un array. ¿Qué pasa?
4	Sumar los primeros diez números usando el ciclo while.
5	Sumar los diez primeros números pares y los diez primeros impares utilizando una única estructura de ciclo.

*04 –

Clases

Una clase define la forma y el comportamiento de los objetos que crea, llamados instancias. La clase es la unidad mínima de código válida que requiere Java, aquí no existe código fuera de una de ellas. En este Capítulo veremos cómo se crean clases, cómo se usan y cómo se definen atributos y métodos. También conoceremos los constructores y elementos estáticos.

- **Definición** 76
- **Atributos** 79
- **Métodos** 82
  - La herencia y los métodos 84
- **Constructores** 85
- **This y Super** 87
- **Lo estático versus lo no estático** 90
- **Resumen** 91
- **Actividades** 92

Definición

Ya hemos visto cómo aparece una clase en nuestros unit tests, ahora entenderemos en profundidad cada elemento de su definición. Todo archivo fuente Java requiere que exista una clase pública con el mismo nombre. A continuación tenemos el caso más simple de una clase:

```
public class Auto {
    ...
}
```

Indicamos que es pública con el modificador **public**, luego viene el **class** y finalmente el nombre. Por defecto, toda clase hereda de **Object**, pero si queremos heredar de alguna otra, podemos especificarlo a continuación del nombre utilizando la palabra **extends** seguida del nombre de la clase padre.

```
public class Auto extends Vehiculo {
    ...
}
```

Solamente se puede heredar de una clase, recordemos que Java implementa herencia simple. Para tratar de alivianar esta restricción, Java utiliza las interfaces para definir protocolos homogéneos a través de jerarquías heterogéneas. Las veremos más adelante (en el capítulo 6), pero para decir que una clase implementa una interfaz (o varias), escribimos el nombre de esta (en el caso de ser varias, separados por comas) después de la palabra **implements**.



WIZARD DE ECLIPSE

Siempre que queramos crear una clase nos conviene utilizar el botón **New...**. O hacerlo desde el menú de Eclipse, ya que allí nos encontraremos con un paso a paso que nos facilitará la correcta creación de la clase. En él tendremos acceso a configurar la nueva clase a gusto, sin escribir una línea de código.

```
public class Auto extends Vehiculo implements VehiculoTerrestre {
    ...
}
```

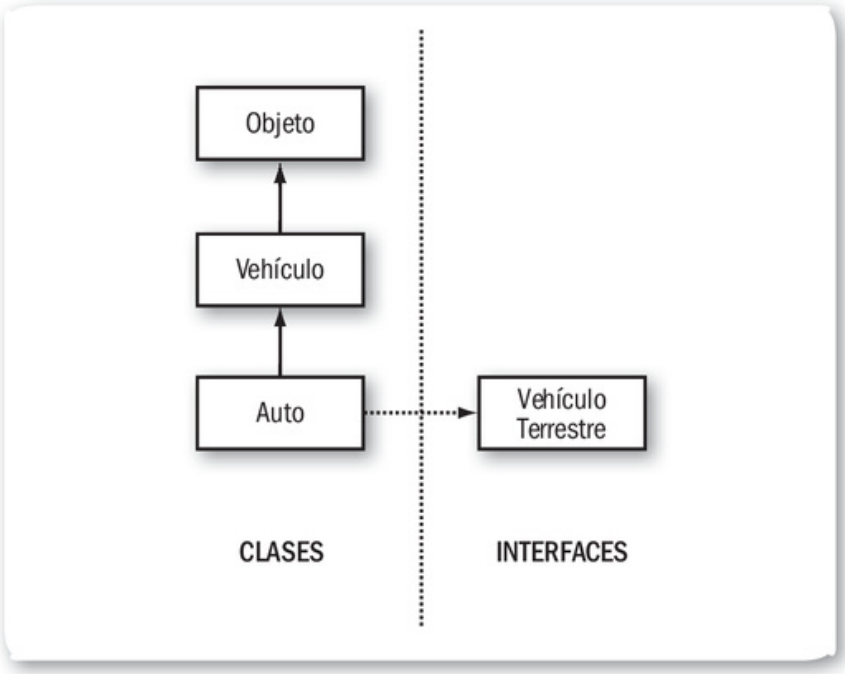


Figura 1. En esta imagen podemos ver una representación que corresponde a la relación de herencia entre las clases y la de implementación con la interfaz.

La clase principal del archivo debe ser pública o no debe indicarse nada (visibilidad por defecto, llamada de paquete), no puede ser privada. Las clases con la visibilidad de paquete solamente pueden ser utilizadas por otras clases que estén dentro del mismo paquete. Las clases públicas pueden ser utilizadas por cualquier otra clase y también pueden ser marcadas como finales, utilizando el modificador **final** antes del **class** en la definición. Este modificador hace que no pueda existir una subclase de ella. Finalmente, las clases pueden ser marcadas como abstractas (moldes para otras clases) usando el modificador **abstract**. Las clases abstractas, como veremos más adelante en detalle, no pueden tener instancias y sirven para agrupar el conocimiento y comportamiento en común de las subclases de este tipo.

Tengamos en cuenta que en Java las clases son instancia de la clase **Class**, en ella están definidos los mensajes que entiende cada clase, como el tipo de mensajes que entienden sus instancias o qué constructores tienen, entre muchas otras cosas.

Para acceder a la instancia de **Class** correspondiente a una clase podemos utilizar el nombre seguido de **.class** o podemos pedírsela a una instancia mediante el mensaje **getClass()**. Ambas formas devuelven un objeto de tipo **Class** que es el dato que nos interesa.

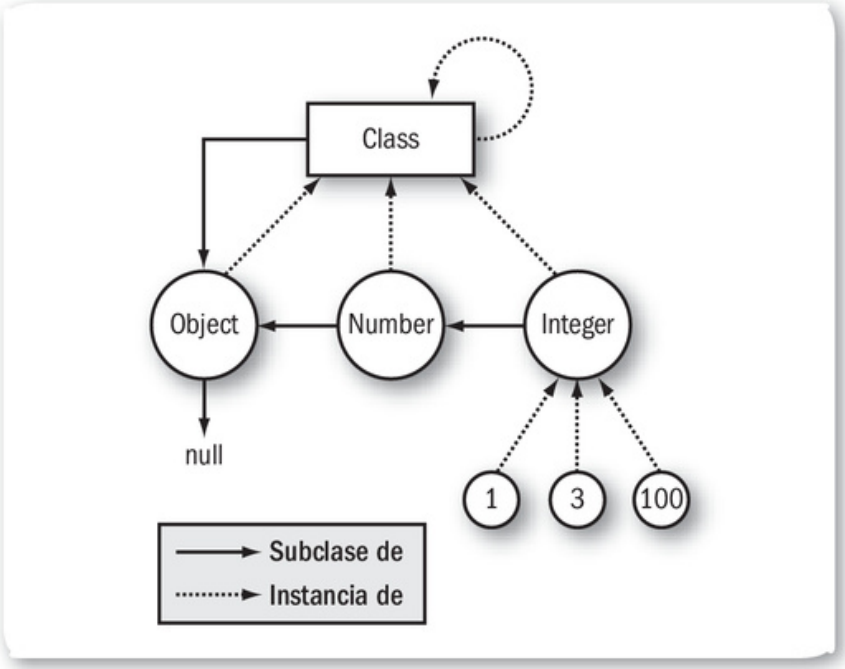


Figura 2. En esta imagen podemos ver una representación de las relaciones que existen entre las clases, la clase denominada `Class` y también las instancias.

PROPIEDADES Y BEANS

Es interesante tener en cuenta que cuando un objeto tiene un **getter** y también un **setter** públicos para un determinado atributo, podemos decir que tiene una propiedad. Llamaremos a la propiedad por el nombre del atributo. Esta es una convención nacida de los **Java Beans** (granos), objetos destinados a ser manipulados gráficamente que nunca cumplieron su objetivo.

Atributos

Los colaboradores internos, también conocidos como variables de instancia o atributos, se definen en el cuerpo de la clase. Su definición es similar a la de cualquier variable, con la particularidad que se le puede agregar los modificadores de visibilidad. Siempre definamos los atributos como privados, dado que son detalles de implementación que deberían estar siempre ocultos a los demás objetos (incluso a aquellos pertenecientes a alguna subclase). Cuando queremos exponer un atributo como público, deberíamos hacerlo a través de métodos **getter** (lectura) y **setter** (escritura).

> LAS VARIABLES DE INSTANCIA O ATRIBUTOS SE DEFINEN EN EL CUERPO DE LA CLASE

Existe la convención de nombrar a estos métodos utilizando los prefijos **get**, **set** o **is**, seguido del atributo.

```
public class Auto {
    private Marca marca;
    ...
    public Marca getMarca() {
        return marca;
}
    public void setMarca(Marca marca) {
    this.marca = marca;
}
    ...
}
```

GETTERS Y SETTERS

En general es una buena idea seguir cada una de las convenciones adoptadas por una comunidad, en este caso utilizar los getters y los setters para acceder a atributos privados. Pero es mejor lograr una claridad de intensión y de lectura que una convención. Pensemos los nombres de los métodos con cuidado, ya que re⊠ejan nuestra intensión a la hora de programarlos.

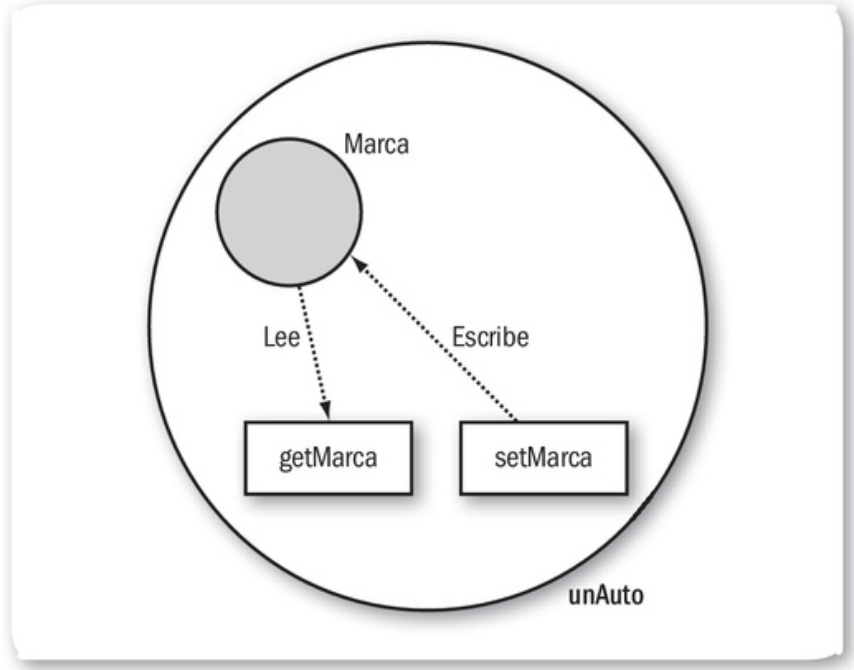


Figura 3. En este diagrama podemos ver la representación de la forma en que se relacionan los **`getters`** y los **`setters`** con el atributo que les corresponde.

Las variables tipo **boolean** utilizan el prefijo **is** en vez del **get**.

```
public boolean isEmpty() {...}
public void setEmpty(boolean isEmpty) {...}
```

Esta es sólo una convención Java, pero no quiere decir que sea obligatoria. Reflexionemos acerca del significado de un método que setea si un objeto esta vacío o no. ¿No tendría que ser, por ejemplo, **vaciar()**? ¿Y lo contrario, **llenar()**? Siempre pensemos en el significado que queremos darle a los métodos en el contexto de uso. Lo importante, es que siempre utilicemos métodos para acceder a los atributos de un objeto, ya sea desde el exterior, como del interior. Esto nos da un grado de flexibilidad muy importante.

Supongamos que tenemos un objeto **Empleado** cuyo sueldo está representado por el atributo público **sueldo**. Cada vez que queramos obtener un dato del tipo **sueldo** estaremos accediendo al atributo.

Ahora supongamos que tenemos un cambio, y el sueldo, en vez de ser un valor fijo, depende de ciertos porcentajes variables. El sueldo tiene que ser calculado cada vez que se lo quiere leer. Tendremos que cambiar absolutamente todos los lugares donde se estaba leyendo el atributo **sueldo** por el envío de mensaje **sueldo()**. No solamente por la adaptabilidad al cambio, sino también por la flexibilidad que da enviar un mensaje (ya que se decide en **runtime** por **method lookup**), es que debemos siempre enviar mensajes en vez de acceder directamente a un atributo, incluso dentro del código de la misma clase.

Las variables pueden ser inicializadas junto con su definición.

```
public class Auto {
    private Marca marca; // inicializado al valor por defecto (null)
    private String modelo = \"\"; // inicializado
    ...
}
```

En los métodos de la clase, estos atributos son accedidos directamente con su nombre o utilizando el pseudo variable **this** que es una referencia al objeto actual.

Los atributos estáticos, aquellos que pertenecen a la clase, se definen agregando el modificador **static**. Estos atributos pueden ser accedidos por medio de su nombre o utilizando el nombre de la clase más el nombre del atributo.

> LOS ATRIBUTOS ESTÁTICOS DEBEN DEFINIRSE AGREGANDO EL MODIFICADOR STATIC

Cuando heredamos de una clase, heredamos los atributos definidos por ella, aunque no podamos acceder a ellos ya que podrían estar declarados como privados. Si estamos creando una jerarquía, es una buena práctica no definir atributos hasta que no lleguemos a las clases concretas (opuesto a lo abstracto; si consideramos que las clases abstractas están en lo alto de la jerarquía, las clases concretas estarían abajo). Si lo hiciéramos estaríamos forzando una implementación particular y concreta en vez de una necesidad abstracta. Deberíamos declarar métodos abstractos (**protected** o **public**, dependiendo de la finalidad). Así las clases concretas pueden decidir ellas mismas que implementación quieren y necesitan.

Métodos

Los métodos son la implementación asociada a un mensaje que entiende el objeto. Su definición consta de una firma que sirve para identificar al método, seguido del código. La firma o signatura de un método se forma con los modificadores de visibilidad, seguido de otros posibles modificadores (**final**, **native**, **synchronized**, **static** y etcétera); luego viene el tipo de respuesta (o **void** si no responde nada), el nombre (que sigue las reglas de los nombres de variables) y, finalmente, los argumentos que espera. Los argumentos se definen como variables (tipo seguido del nombre) con la posibilidad de aplicar el modificador **final**.

```
class Auto {
...
public Marca getMarca() {
   return this.marca;
}
public void arrancarCon(final Llave laLlave) throws LlaveIncorrectaException
{...}
...
}
```

Si un método lanza excepciones, ya sea porque su propio código las lanza o no las atrapa, debe especificarlas como parte de la firma del mensaje e indicarlas luego de los argumentos utilizando la palabra **throws** seguido de los nombres de estas separados por coma.

Java permite que los mensajes tengan argumentos variables, por ejemplo si queremos pasar un cierto número de parámetros en un envío y cierto número en otro, y queremos que además sea invocado el mismo método. En Java podemos hacer esto así:

```
void cargarEquipaje(Equipaje ... equipajes) {
...
}
```

Y el envío de mensaje se realiza de la siguiente forma.

```
...
auto.cargarEquipaje(unEquipaje, otroEquipaje, yOtroEquipajeMas);
...
```

Recordemos que se coloca la cantidad deseada de parámetros separados por comas. Esto es en realidad una facilidad del compilador que transforma el código de la siguiente forma:

```
void cargarEquipaje(Equipaje [] equipajes) {
...
}
...
auto.cargarEquipaje(new Equipaje[] {unEquipaje, otroEquipaje,
    yOtroEquipajeMas});
...
```

Sí, transforma todo del mismo modo que si el mensaje recibiera un **array** y en el código del método accediéramos a los argumentos utilizando el parámetro como un array.

```
void cargarEquipaje(Equipaje ... equipajes) {
    for(Equipaje equipaje : equipajes) {
        this.cargar(equipaje);
        }
}
```



USO DEL FINAL

Siempre debemos tener presente que se trata de una buena práctica el modificar tanto los argumentos de los métodos como las variables que definimos en ellos con **final**. De esta forma podremos evitar cometer errores al reasignarles valores a tales variables (o también argumentos) en el método que corresponda. Estos errores son generalmente difíciles de detectar.

Java permite que se puedan definir varios métodos con el mismo nombre dentro de una misma clase. Esto se conoce como sobrecarga (**overloading**). La restricción que existe es que lo métodos difieran en la cantidad o tipo de argumentos que reciben.

```
void acelerarA(int kmPorH) {...} // válido

void acelerarA(double kmPorH) {...} // válido

void acelerarA(int km, int h) {...} // válido

bool acelerarA(int kmPorH) {...} // inválido
```

El compilador se encarga de decidir cuál mensaje se envía en base a la información de tipos y de cantidad de parámetros que se pasan.

La herencia y los métodos

Cuando heredamos de una clase, nuestros objetos entienden todos los mensajes declarados en ella y en toda la ascendencia de clases. A medida que vamos extendiendo la jerarquía, ampliamos el conocimiento, agregando nuevo comportamiento, o especializamos el comportamiento heredado. Agregar nuevo comportamiento es simplemente definir nuevos mensajes al agregar nuevos métodos. En cambio, cuando queremos especializar cierto comportamiento, lo que buscamos es que ciertos objetos se comporten de forma distinta que los objetos de clases padres al recibir el mismo mensaje, lo que hacemos es redefinir el mismo método (con la misma firma) en la clase que nos interesa.



USO CORRECTO DEL SUPER

Es interesante saber que cuando usamos **super** no tenemos restricciones sobre qué mensajes enviamos. Por esta razón una buena práctica que debemos adoptar es restringir los envíos con **super** al mismo mensaje que se está ejecutando. Así es más fácil seguir el código para nosotros y para otros. De este modo cometeremos menos errores difíciles de detectar y corregir.

```
public class Lista {
    public void agregar(Object objeto) {...}
}

public class ListaOrdenada extends Lista {
    @Override
    public void agregar(Object objeto) {
        super.agregar(objecto);
        this.ordenar();
    }
}
```

La anotación **@Override** es opcional, pero es una buena práctica agregarla, así al compilar el IDE pueden avisarnos si no estamos redefiniendo un método porque nos equivocamos en la firma. Como buena práctica, deberíamos siempre utilizar **super** para invocar el comportamiento que estamos especializando, generalmente como el primer paso de un método o como el último. Si lo llamamos en medio de la ejecución, es porque queremos hacer algunos preparativos antes de llamar y luego queremos procesar el resultado antes de devolverlo.



Constructores

Los constructores son métodos especiales que sirven para inicializar las instancias de una clase. Son métodos que se llaman igual que la clase, no tienen tipo de retorno (ni siquiera **void**) y solamente se les pueden aplicar los modificadores de visibilidad.

```
public class Celular extends Telefono {
...
    public Celular() {...}
    public Celular(NumeroTelefonico numero)
    throws NumeroTelefonicoInvalido {...}
    public Celular(
```

```
        Prefijo prefijo,
            NumeroTelefonicoLocal numeroLocal
        ) throws NumeroTelefonicoInvalido {...}
        ...
}
```

Los constructores también se pueden sobrecargar, como cualquier método, variando la cantidad de parámetros que reciben.

Tengamos en cuenta que los constructores son invocados (no es un envío de mensaje) cuando se crea una instancia. Normalmente crearemos instancias mediante el operador **new**.

```
Utensilio tenedor = new Tenedor();
```

El operador new se encarga de crear una instancia de la clase; reserva espacio en la memoria para la instancia, para todos sus atributos. Luego con la instancia creada, se invoca al constructor para que la inicialice. Java crea automáticamente un constructor sin parámetros y público, en caso que el programador no haya declarado algún constructor en la clase correspondiente. Una clase padre fuerza los constructores en sus clases hijas. Por eso una clase hija tiene que definir un constructor con la misma firma que el de la clase padre. Además cuando una clase define un constructor, debe invocar algún constructor de la clase padre como primer paso en el código. Esto es para asegurarse de que lo definido por la clase padre está bien inicializado antes de inicializar los atributos declarados por la clase hija. Esto lo lograremos usando la palabra **super** (y los argumentos necesarios) al llamar al constructor padre.



CONSTRUIR OBJETOS VÁLIDOS

Debemos tener en cuenta que es conveniente realizar la inicialización de los objetos mediante un constructor en vez de utilizar getters. De esta forma podemos asegurarnos de que estamos construyendo objetos válidos para el modelo. Si no, debemos arrojar alguna excepción para notificar el error. La excepción a esto es cuando se requieren muchos parámetros.

```
public class Telefono {
public Telefono(NumeroTelefonico numero) {...}
    ...
}
...
public class Celular extends Telefono {
    public Celular(NumeroTelefonico numero) {
        super(numero);
        ...
    }
    ...
}
```

Recordemos que las clases hijas pueden definir otros constructores además de los establecidos por la clase padre.

This y super

Debemos tener en cuenta que en los métodos de instancia (así como también en los constructores correspondientes) tenemos a nuestra disposición dos pseudo variables (se llaman así porque no se encuentran definidas en ningún lugar), estas son **this** y **super**; **this** es una referencia al receptor del mensaje cuyo método está siendo ejecutado. Con **this** tendremos acceso a todos los elementos del objeto, atributos, métodos y constructores. Recordemos que el uso de **this** es opcional salvo cuando tenemos que desambiguar algún nombre.



SIEMPRE LLAMAR A SUPER

Sabemos que es requerido llamar a algún constructor de la clase padre siempre, salvo cuando no heredamos de nada, ya que heredamos de **Object** y de esta forma el compilador lo hace por nosotros. Si nos encontramos con este caso es recomendable hacerlo igual. Así nos evitaremos problemas en el momento que decidamos hacer heredar la clase de alguna otra.

```
public class Celular extend Telefono {
    private Prefijo prefijo;
    ...
    public Celular(NumeroTelefonico numero) {
        // utilizo this para llamar al otro constructor
        this(numero.prefijo(), numero.local());
    }

    public Celular(
        Prefijo prefijo,
        NumeroTelefonicoLocal numeroLocal
    ) {
        // llamo al constructor de la clase padre
        super();
        // utilizo this para indicar que accedo
        // al prefijo de la instancia y no al argumento
        this.prefijo = prefijo;

        ...
    }
    ...
    public Prefijo getPrefijo() {
        return this.prefijo; // acá el this es opcional
    }
}
```

super, al igual que **this**, es una referencia al objeto receptor del mensaje cuyo método está siendo ejecutado. La diferencia con el



SUPER

Mucha gente (la mayoría) confunde el verdadero significado de **super**. En general, la mayoría de los programadores creen que super significa la superclase, la superclase del objeto receptor. Este error de concepto no impacta en la mayoría de los casos, pero en escenarios donde las jerarquías son complicadas, y se sobrescriben muchos métodos, puede llevar a errores difíciles de encontrar.

anterior radica en que no se puede utilizar salvo para enviar mensajes y, lo más importante, que los métodos son buscados a partir de la clase padre de la clase a la que pertenece el método que se está ejecutando. Releamos la definición anterior una vez más ya que es muy importante y en la mayoría de los lugares no está bien definida. El siguiente código ejemplifica el uso y funcionamiento de **super**.

```
public class A {
    public String bar() {
        return "A";
    }
    public String zip() {
        return "A";
    }
    public A getThis() {
        return this;
    }
}
...
public class B extends A {
    public String foo() {
        return super.bar();
    }
    @Override
    public String bar() {
        return "B y " + super.bar();
    }
}
...
public class C extends B {
    @Override
    public String bar() {
        return "C y " + super.bar();
    }
    @Override
    public String zip() {
        return "C y " + super.zip();
```

```
    }
}
...
public class SuperUnitTest {
    @Test public void testSuper() {
        C c = new C();
        assertSame(c, c.getThis());
        assertEquals("A", c.foo());
        assertEquals("C y B y A", c.bar());
        assertEquals("C y A", c.zip());
    }
}
```



Lo estático versus lo no estático

Los métodos y atributos que definimos como estáticos pueden parecer que pertenecen a la clase, ya que no pertenecen a las instancias, y los accedemos mediante el nombre de la clase. Si recordamos que todas las clases son instancias de la clase **Class**, entenderemos que todas ellas responden a los mismos mensajes y poseen los mismos atributos. Los elementos estáticos están asociados a la clase, pero no pertenecen a ella; el compilador, junto con la **JVM,** manejan el acceso a ellos. Estos elementos son globales. Los métodos estáticos son resueltos en tiempo de compilación y no resultan del envío de un mensaje. Es posible acceder a los métodos estáticos de una clase padre usando el nombre de una clase hija. Debemos saber que en ellos no se usa **this** ni **super**.

En cambio, los métodos de instancia son resueltos en tiempo de ejecución y resultan del envío de un mensaje. Esto da una flexibilidad que no se tiene con los métodos estáticos ya que el comportamiento está dado por cómo relacionamos los objetos durante la ejecución, en vez de forzarlo en tiempo de compilación a un código específico. Los envíos de mensajes son puntos de acceso a distintos comportamientos, ya que el objeto receptor puede cambiar. En los métodos estáticos está fijo para siempre, a menos que cambiemos el código fuente y lo recompilemos.

Por eso la recomendación siempre es tratar con objetos y dejar de lado lo estático. Al hacer esto estamos siendo más tolerantes al cambio. Recordemos que los cambios siempre existen.

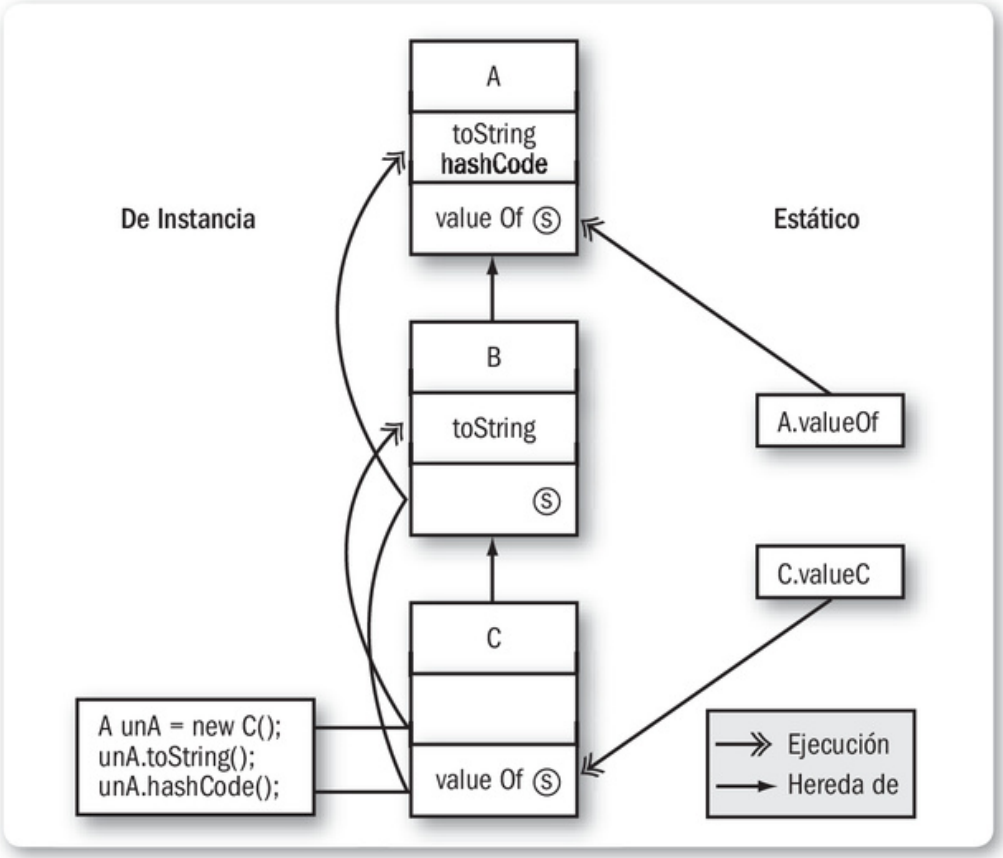


Figura 4. Este diagrama nos presenta la diferencia entre ejecutar un método estático y realizar el envío de un mensaje.

RESUMEN

Este es un capítulo muy importante ya que presenta la forma principal de Java de catalogar comportamiento y la única forma que tiene el lenguaje para implementar código. Todo código en Java existe dentro de una clase, ya sea estático o de instancia. Hemos aprendido como definir atributos y métodos, también qué significa la pseudo variable **this** y la diferencia con **super**. Vimos la diferencia entre el alcance de instancia y el estático, y los motivos para alejarnos de este último. Por último empezamos a ver como se hereda comportamiento de otra clase y cómo formar una jerarquía de tipos.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿De cuántas clases se puede heredar?
2	¿Cuántas interfaces puede implementar una clase?
3	¿Qué es un atributo?
4	¿Qué son los **`getters`** y los **`setters`**?
5	¿Qué es **`this`**?
6	¿Qué es **`super`**?
7	¿Cómo funcionan los argumentos variables?
8	¿Cuáles son las características de los constructores?
9	Enumere algunas diferencias entre el alcance estático y el alcance de instancia.
10	¿Existe un **`this`** cuando estamos en el alcance estático?

ACTIVIDADES PRÁCTICAS

1	Crear una clase.
2	Definir un atributo y sus métodos **`accessors`**.
3	Redefinir el método **`toString`** (definido en Object).
4	Agregarle un método a la clase y sobrecargar el método agregando nuevos parámetros.
5	Crear una subclase y sobrescribir un método de la clase padre utilizando **`super`**.

*05 –

Más Clases

En este capítulo introduciremos conceptos más avanzados sobre las clases y su uso. Abarcaremos las nociones de clase abstracta, clase anidada y, finalmente, clase anónima. La clase abstracta representa un molde para otras clases, al agrupar conocimiento para formar una jerarquía. También enfrentaremos la generación del proyecto: El juego de la vida.

- Clases abstractas 94
- Clases anidadas 97
- Clases locales y clases anónimas 101
- Ejercicio: El juego de la vida 103
- Resumen 121
- Actividades 122

Clases abstractas

En el capítulo anterior aprendimos la utilización básica de las clases en Java. Vimos cómo se definen, cómo se declaran los atributos, métodos y constructores. Conocimos y entendimos la diferencia que existe entre utilizar atributos y métodos de instancia frente a los estáticos, y la razón por la cual es importante no utilizarlos.

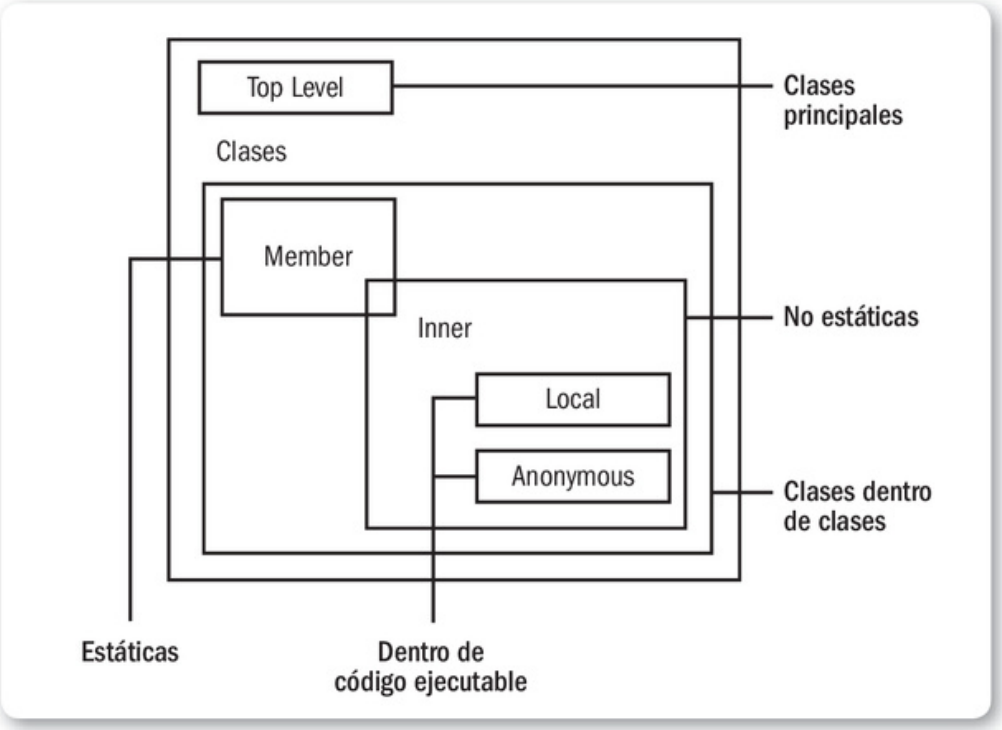


Figura 1. En esta imagen podemos ver un esquema con los distintos tipos de clases y cómo se relacionan entre sí.

Visualicemos la taxonomía de los animales, tenemos los que son mamíferos y los que no. Dentro de la categoría de los mamíferos nos encontramos con los caninos y, a su vez, dentro de los caninos, tenemos a los perros. En esta clasificación "mamíferos" y "caninos" son categorías que agrupan animales que poseen ciertas características comunes entre sí. Las clases abstractas son el equivalente a estas categorías y ellas definen el comportamiento esperado para un conjunto de objetos. Estos objetos pertenecen a una subclasificación de la clase abstracta, ya que una clase abstracta no puede tener instancias.

Pensémoslo de este modo: no existe un mamífero que no pertenezca a alguna subclasificación, existen perros que son caninos que a su vez son mamíferos. No existe un animal al que llamemos mamífero. Pero debemos tener en cuenta que si tenemos un perro, este tiene todas las características que lo definen como mamífero.

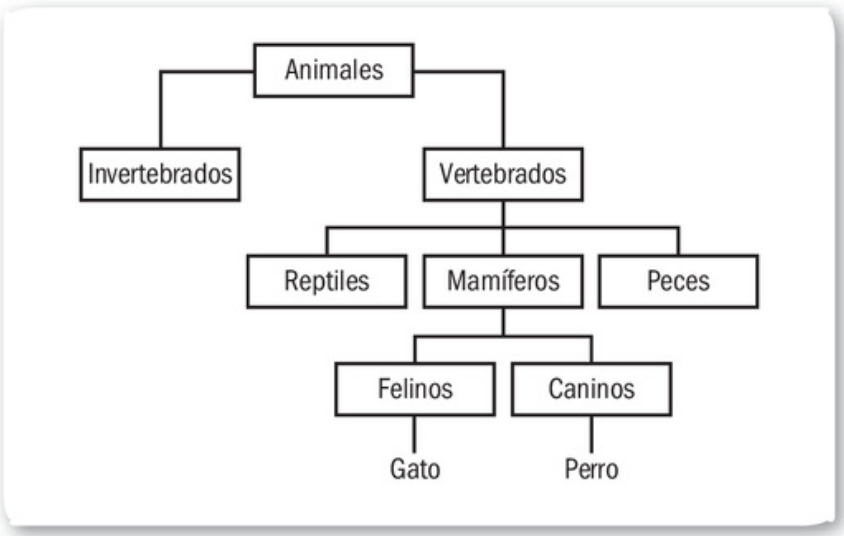


Figura 2. En esta imagen podemos ver una posible categorización de los animales (incompleta).

Tengamos en cuenta que las clases abstractas en Java se declaran agregando el modificador **abstract** a la definición.

```
public abstract class Mamifero {
    ...
}
```

Las clases abstractas pueden especificar métodos sin definirles código. Estos son los llamados métodos abstractos, que declaran un comportamiento esperado para los objetos que sean de ese tipo y dejan la implementación específica a las subclases. Debemos saber que los métodos abstractos se definen con el modificador **abstract**.

Estos métodos son un punto de extensión explicito ya que indicamos que van a existir subclases con un comportamiento distinto.

```
public abstract class Mamifero {
    abstract void amamantarA(Mamifero unaCria);
}
```

Las clases abstractas aparecen porque notamos que algunas de ellas tienen comportamiento en común y pueden ser agrupadas bajo un mismo tipo. Compartir código no es razón suficiente para crear una clase abstracta y hacer que esas clases hereden de la superior. La razón tiene que ser que la clase abstracta agrupa a todas ellas bajo un mismo concepto. Lo que se busca es reutilizar el conocimiento. La reutilización de código es una consecuencia, no la causa. Una jerarquía correctamente diseñada forma un árbol, con **Object** como raíz. Las ramas se forman con clases abstractas y solamente las hojas deberían ser clases concretas (no abstractas). El motivo es hacer que las clases padre fuercen detalles de implementación a sus clases hijas, principalmente en la forma de atributos. Sólo se deberían definir métodos con implementación y métodos abstractos. También es necesario tener presente que debemos buscar que en vez de definir métodos abstractos se definan métodos con una implementación por defecto. De esta manera las clases hijas sólo implementan lo que realmente desean y no se ven forzadas a implementar otros métodos.

> UNA JERARQUÍA CORRECTAMENTE DISEÑADA FORMA UN ÁRBOL, CON OBJECT COMO RAÍZ

Las clases abstractas, por definición, no pueden ser marcadas como final, ya que su propósito es que sirvan como base para otras clases.

Recordemos que el principio de ocultamiento de la información vale también entre clases de la misma jerarquía.



CLASES DIOS

En la jerga de la orientación a objetos, se dice que una clase en una clase dios cuando tiene muchos atributos, muchos métodos. Generalmente estos métodos responden a distintas responsabilidades. Estas clases deben ser refactorizadas inmediatamente, y separarlas en varias clases más chicas que modelen correctamente las responsabilidades por separado.

Clases anidadas

Las clases anidadas son las que se definen en el contexto de otra clase. Existen cuatro tipos: las estáticas, las internas a la clase, las internas a los métodos y, finalmente, las anónimas. Salvo las estáticas, las demás son para uso interno de la clase en la que se definen. Generalmente se utilizan para modelar conceptos que están fuertemente ligados a una clase y que no tienen sentido fuera de ella, ya que su uso es en conjunto.

Clases anidadas estáticas

Conocidas como **static nested classes**, son clases definidas dentro de otra, pero son independientes de la clase contenedora. Por lo tanto pueden ser públicas y ser accedidas y usadas desde otras clases. El acceso a ellas es a través de la clase contenedora. Para declararlas, definimos una clase dentro de la clase contenedora y la marcamos como estática con el modificador **static**.

```
public class Auto {
    public static class Llave {
    ...
    }
...
}
```

En nuestro ejemplo la clase interna **Llave** es accedida desde otras clases utilizando el nombre **Auto.Llave**, que deja en claro la relación que existe entre las clases **Auto** y **Llave**. Supongamos que también definimos la clase **Lancha** y su correspondiente clase estática **Llave**. Cuando hablamos de **Llave** en el contexto de **Lancha** hacemos referencia a la clase

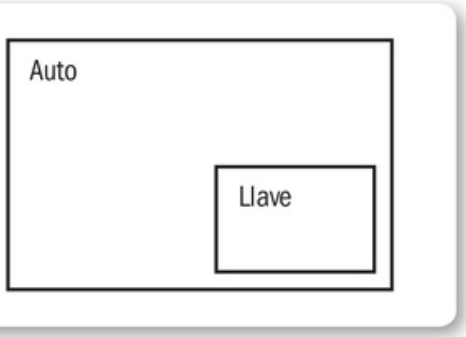


Figura 3. Diagrama de la clase Auto y su clase anidada Llave.

asociada a esta, cuando hablamos en el contexto de **Auto** queremos hablar de **Llave** con una asociación distinta. Las clases estáticas sirven para este propósito, ya que si bien ambas tienen el mismo nombre, al tener otro contexto son dos clases totalmente diferentes.

Clases internas

Llamadas **inner classes**, al igual que las estáticas son clases definidas dentro de otra, aunque son diferentes de estas ya que sus instancias siempre están ligadas a una instancia de la clase contenedora. Su definición es igual a la ya vista, sin utilizar el **modificar static**. Debemos tener en cuenta que, en general, las clases internas son exclusivamente de uso interno de la clase, por lo que casi nunca son públicas.

```
public class Auto {
    public class Freno {
        ...
    }
    ...
}
```

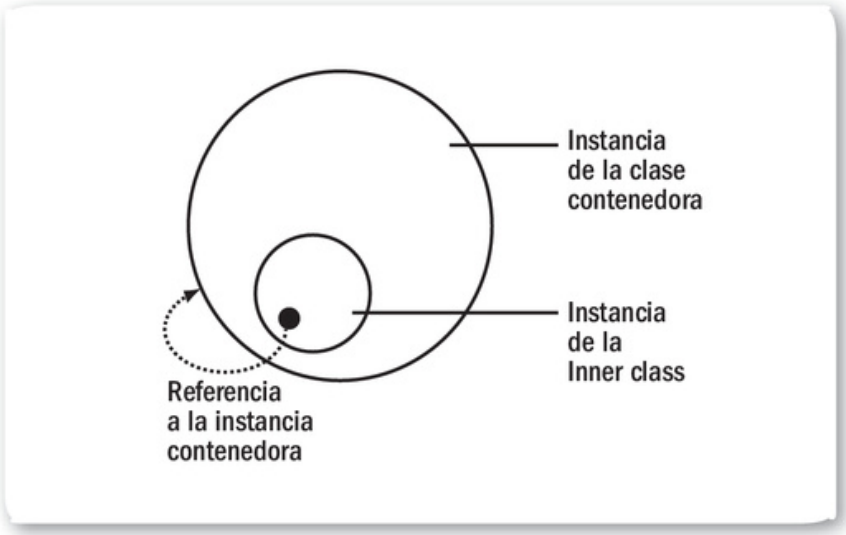


Figura 4. Relación entre una instancia de una `inner class` y la instancia contenedora.

Existen tres grandes diferencias entre las **static inner clases** y las **nested classes**. La primera es que las clases internas son parte de la definición de la clase (como un método de instancia) y por lo tanto pueden acceder a todos los elementos de esta, ya sea atributos o métodos, incluso privados. La segunda es la forma de instanciarlas. En los métodos de instancia de la clase contenedora no hay diferencia, pero si la **nested class** es pública, necesitamos de una instancia de la clase contenedora.

```
public class AutoUnitTests {
    ...
    @Test public void testFrenoInnerClass() {
        Auto unAuto = new Auto();
        Auto.Freno unFreno = auto.new Freno();
    }
    ...
}
```

Notemos cómo debemos utilizar el operador **new** junto con la instancia de la clase contenedora. La tercera diferencia radica que dentro de los métodos de instancia de la **inner class** tenemos accesos, no solo de referencia a la instancia correspondiente de la clase (utilizando el **this**), sino también a la instancia asociada de la clase contenedora.

```
public class Auto {
    class Freno {
        public void frenar()  {
            Auto.this.frenar();
```



COPIAS DEFENSIVAS

Recordemos que cuando retornamos algún objeto que sea colaborador interno, debemos tener cuidado de que este no pueda ser modificado sin conocimiento del objeto que lo contiene. Generalmente se devuelve una copia si el objeto es mutable. Las colecciones son candidatas naturales a la copia. Recordemos que los asuntos internos de un objeto no deben filtrase al exterior.

```
        // envía frenar al objeto de tipo Auto asociado
        }
    }
...
}
```

Como vemos en el ejemplo, utilizamos el nombre de la clase contenedora y **this** para la referencia. Se preguntarán cómo se realiza la asociación, esta se realiza automáticamente cuando se instancia un objeto de la clase anidada en los métodos de instancia de la clase contenedora. Se asocia al receptor del mensaje que se ejecuta.

```
public class Auto {
    class Freno {
        public void frenar()  {
            Auto.this.frenar();
        }
    }
    public Auto() {
        this.setFreno(new Freno());
    // asociación automática
}
public void frenar() {
    ...
}
...
}
```



USO DE LOS STATIC IMPORTS

Es necesario tener en cuenta que el uso de los **static imports** debe tener como objetivo fundamental realizar la aclaración de las intenciones del programador cuando escribe código. Además, deben estar limitados solamente al uso de static factory methods**,** ya que de lo contrario estaríamos encapsulando colaboraciones interesantes en un método estático, perdiendo ⊠exibilidad.

Clases locales y clases anónimas

Las clases locales son clases internas que definen los métodos de instancia de la clase contenedora. Están confinadas al método donde están definidas, por lo que no debemos especificar la visibilidad. Mantienen las propiedades de las **inner classes** y pueden acceder a parámetros y variables locales de método si están marcadas como **final**.

```
public class RelojDigital extends Reloj {
    ...
    public Tiempo ahora() {
        final Hora hora = horaActual();

        class TiempoDeRelojDigital extends Tiempo {
            @Override
            public Hora hora() {
                return hora;
            }
            @Override
            public Minutos minutos() {
                return RelojDigital.this.minutosActuales();
            }
        }
        return new TiempoDeRelojDigital();
    }
    ...
}
```



SOBRE LAS CLAUSURAS

Una clausura es una función en conjunto con su entorno, donde las variables libres están ligadas al entorno donde ésta defina la función. Las clausuras se utilizan para definir estructuras de control, encapsular estado y eventos. Java no soporta este concepto, lo más cercano son las clases anónimas.

Las clases anónimas o **anonymous classes** son aquellas que son locales, que no tienen nombre y sólo existen para especificar el comportamiento de la única instancia que tienen. Estas clases son muy útiles para implementar mecanismos de **callback** (de aviso y respuesta asincrónica). A estas clases no se les puede definir un constructor, pero sí tienen un bloque de código inicializador que se ejecuta después de creada la instancia. Sin embargo, se pueden utilizar los constructores de la superclase para crear la instancia anónima.

```
abstract class ObservacionSobre {
    public ObservacionSobre(Object unObjecto) { ... }
}
public class ObservacionSobreUnitTests {
    @Test public void testInicializacion() {
        Mariposa unaMariposa = new Mariposa();
        // creamos una instancia usando el constructor
        ObservacionSobre observación = new ObservacionSobre(unaMariposa) {
            { ... } // código inicializador
            ...
            // se pueden sobrescribir métodos
            // y crear nuevos, es una clase
            // normal en ese sentido
        };
    }
}
```

Las clases anónimas tienen las mismas características que las clases locales, por lo tanto pueden acceder a parámetros y variables locales finales del método como a la instancia de la clase contenedora.



INCODE ECLIPSE IDE PLUGIN

Si apuntamos nuestro navegador a la siguiente dirección **http://loose.upt.ro/incode/pmwiki.php/Main/InCode**, encontraremos un plugin para Eclipse que analiza nuestro código y genera métricas sobre nuestro diseño. Frente a malos diseños, el plugin nos dará consejos sobre cómo modificarlo para solucionarlo.

Ejercicio: El juego de la vida

Ya disponemos de las herramientas y los conocimientos necesarios para crear programas complejos e interesantes.

El juego de la vida es un autómata celular ideado por el matemático británico **John Horton Conway** en 1970. Es un juego sin jugadores, ya que, dado el estado inicial, las reglas hacen que el juego evolucione solo.

El juego consiste en una grilla formada por celdas cuadradas donde

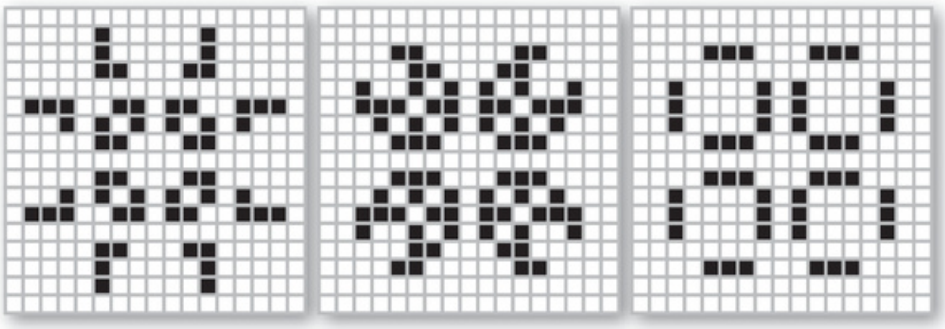


Figura 5. El patrón **`pulsar`**, se trata de un patrón oscilante que tarda cuatro turnos en recuperar su forma original.

cada celda (o célula) puede estar viva o muerta. Las celdas adyacentes se llaman vecinos. Cada célula interactúa con los ocho vecinos que tiene a su alrededor. El juego avanza en turnos y en cada turno se transforma la grilla aplicando las siguientes cuatro reglas:

1) Si una célula viva tiene menos de dos vecinos vivos, muere.
2) Debemos tener en cuenta que si una célula viva tiene dos o tres vecinos, sigue viva durante el siguiente turno.
3) Si una célula viva tiene más de tres vecinos, muere.
4) Si una célula muerta tiene exactamente tres vecinos, revive.

Recordemos que el estado inicial de la grilla se conoce como semilla del sistema. Desde ese punto, las reglas se aplican en simultáneo sobre el estado de la grilla en el turno pasado. Existen varios patrones iniciales conocidos que permiten generar naves que viajan por el tablero, o balizas que titilan o generadores de formas.

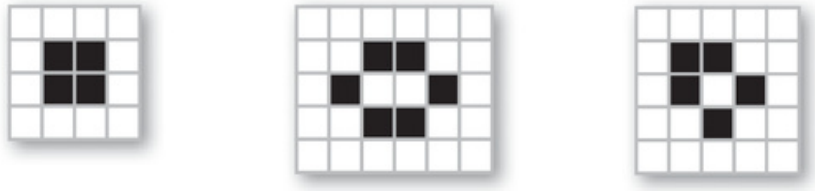


Figura 6. Tres patrones clásicos del juego: **`bloque`**, **`colmena`** y **`bote`**. Estos patrones son estables, no cambian su forma a lo largo de los turnos.

Como siempre, para codificar la solución, necesitaremos un **unit test** que nos guíe en el desarrollo. Este test nos permitirá explorar una API clara para nuestro juego. Empecemos con un caso sencillo, una grilla de cuatro por cuatro con solamente una célula viva. Esta configuración sólo necesita un paso de evolución para que todas las células estén muertas.



Figura 7. Como vemos aquí, una célula viva solitaria muere en un turno. Este es el caso más simple de juego.

Escribimos esto mismo en un **unit test**, sin preocuparnos por los errores que nos indica el IDE.

```
@Test public void testSimpleGame() {
GameOfLife game = newGameWithDimensions(4,4);
game.seedLiveCell(at(1,1));
game.evolveOneStep();
allCellsAreDead(game);
}
```

El test indica que queremos un juego nuevo de dimensiones 4x4, donde queremos que la célula en la posición (1,1) esté viva inicialmente. Luego evolucionamos el juego un turno y verificamos que todas las células del tablero están muertas. Tendremos varios errores, pero no hay que preocuparse ya que los iremos corrigiendo mientras programamos. Definamos los métodos para crear el juego y verificar las células.

```
GameOfLife newGameWithDimensions(int i, int j) {
    return GameOfLife.newGameWithDimensions(i,j);
}
void allCellsAreDead(final GameOfLife game) {
    for(Association<Point, Cell> cellAndPosition :
    game.cellsAndPositions()) {
        Cell cell = cellAndPosition.value();
        assertTrue(cell.isDead());
    }
}
```

Bastante sencillo, primero encapsulamos la creación del juego en un método separado, lo que nos permitirá reutilizar los tests para otras implementaciones del juego y así compararlas. Luego tenemos el acceso a las células y sus posiciones en la grilla. Para no exponer la implementación al cliente, lo mejor es brindar un iterador por los pares (asociaciones) de cada célula con su posición. Tomemos nota de algunas características de este código. Primero, la creación del juego la hacemos a través de un método estático (**static factory method**) en vez de usar directamente el constructor. El motivo, entre otros, es que los constructores no tienen nombres, entonces es difícil saber qué hacen.



STATIC FACTORY METHODS

Para considerar **static factory methods** sobre constructores, primero debemos saber que poseen un nombre que explica la intensión del método. Es importante para tener código claro. Segundo, pueden no crear un objeto nuevo por cada uso. Tercero, pueden devolver objetos de un subtipo en vez del tipo en el que están definidos, y así lograr un grado de ☒exibilidad extra.

Tengamos en cuenta que esta recomendación es conveniente aplicarla en casos donde el constructor acepte varios parámetros o donde haya varias formas distintas de crear el objeto.

Para la inicialización de las células escribimos los métodos que se muestran en el siguiente código:

```
GameOfLife seedLiveCell(final Point seed) {
    return seedLiveCells(seed);
}

GameOfLife seedLiveCells(final Point ... seeds) {
    for (Point seed : seeds) {
        grid().putAt(seed, liveCell());
    }
    return this;
}
```

El primer método es para agregar de a una célula por vez, que también utiliza el segundo, pues se trata de una extensión para agregar varias simultáneamente. Vemos el uso de argumentos variables y observemos el que presenta **final** en los argumentos. Las células se ubican en una grilla, modelada por la clase **Grid**. Esta clase genérica (la veremos este tema más adelante), se va a encargar de manejar la ubicación de las células por nosotros. Para inicializar la grilla utilizamos el mismo mecanismo que con **GameOfLife**, un static factory method, al que también le indicamos un objeto **Default** para la inicialización de cada celda de la grilla. **GameOfLife** define el valor inicial de la celda de la grilla como una célula muerta.



UTILIZAR LA MÁXIMA ABSTRACCIÓN POSIBLE

Cuando especificamos el tipo de un parámetro o el tipo de retorno de un método debemos pensar siempre en la máxima abstracción que nos sirva. Esta abstracción debe ser el requerimiento mínimo de nuestro método. Utilizaremos en estos casos clases abstractas e interfaces en vez de clases concretas.

```
Grid<Cell> newGrid(int width, int height) {
    return Grid.newOfWidthAndHeightWithDefault(
                width,
                height,
                defaultCell());
}

Default<Cell> defaultCell() {
    return new Default<Cell>() {
        public Cell newOne() {
            return deadCell();
        }
    };
}

Cell deadCell() {
    return new DeadCell();
}
```

Utilizamos una clase anónima para definir el valor por defecto. Notemos que cada conjunto de colaboraciones (envíos de mensajes) que utilizamos para crear algún objeto está encapsulado en un método (con visibilidad **protected**). Esto nos permite modificar los objetos creados en una posible subclase y nos da un alto grado de flexibilidad.

Es necesario que observemos que estamos tratando de ser lo más abstractos posible en las firmas de los métodos, ya que indicamos el retorno de un **Cell** que devuelve un **DeadCell**.

Para facilitarnos el trabajo, modelaremos la grilla como una serie de filas (clase **Row**). De esta forma, el manejo de los índices a las celdas es más sencillo, ya que primero nos preocupamos de las filas y luego el objeto fila se encargará de manejar el acceso a la celda en particular.

Pasemos a la evolución de las células. Este es un buen momento para releer las reglas mencionas al inicio. De ellas se desprende que la vida o muerte de una célula está dictada por su entorno, su vecindario, o sea, por las células que la rodean. También sabemos que solamente puede haber células vivas o muertas y que dependen de lo que sean y del vecindario para su evolución. Definamos, entonces, las células.

```
public abstract class Cell {
    public boolean isDead() {
        return false;
    }

    public boolean isAlive() {
        return !isDead();
    }

    abstract public Cell evolveAccordingTo(
            final Neighborhood neighborhood);

}
```

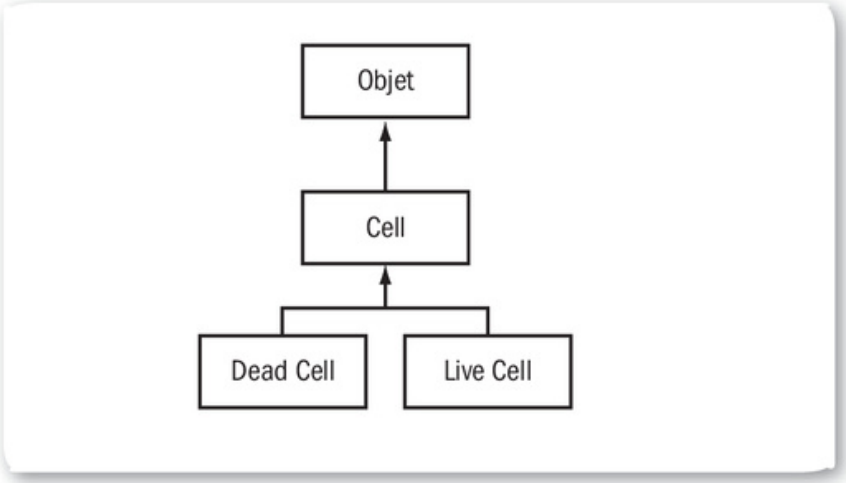


Figura 8. Esta imagen nos muestra una jerarquía de las clases que representan a las células.

MÉTODOS DE CONSULTA DE ESTADO

Cuando necesitamos responder los mensajes de estado de un objeto (generalmente comienzan con el prefijo **is** y retornan un **boolean**) es conveniente poder responder a los opuestos (**isDead** y **isAlive**) e implementar uno en función del otro, para facilitar la lectura y el uso del objeto evitando utilizar la negación.

Cell es la clase abstracta para representar a todas las células. Declaramos que todas las células pueden responder si están vivas o muertas y, también, que su evolución depende de las células a su alrededor. Subtipamos las células en vivas o muertas explícitamente en el diseño al crear clases para ambos tipos.

```
public class DeadCell extends Cell {
    @Override
    public boolean isDead() {
        return true;
    }
}
public class LiveCell extends Cell {
}
```

Es obvio que la implementación del mensaje **isDead** para una célula muerta es simplemente devolver **true**. **LiveCell** hereda directamente el comportamiento de **Cell** en este caso.

Ahora pensemos en cómo modelar la interacción entre el vecindario y las células. En principio hay que contar cuántos vecinos vivos conforman el vecindario. Según las reglas nos interesa contar si hay cero, uno, dos, tres y más de tres vecinos. Modelemos exactamente esto, empezamos con un vecindario vacío. Si agrego una célula viva, paso a tener un vecindario de uno, etcétera.

```
abstract class Neighborhood {
    abstract public Neighborhood receiveLiveCell();
}
class ZeroNeighborsAlive extends Neighborhood {
    @Override
    public Neighborhood receiveLiveCell() {
        return new OneNeighborAlive();
    }
}
class OneNeighborAlive extends Neighborhood {
    @Override
```

```
    public Neighborhood receiveLiveCell() {
        return new TwoNeighborsAlive();
    }
}
class TwoNeighborsAlive extends Neighborhood {
    @Override
    public Neighborhood receiveLiveCell() {
        return new ThreeNeighborsAlive();
    }
}
class ThreeNeighborsAlive extends Neighborhood {
    @Override
    public Neighborhood receiveLiveCell() {
        return new MoreThanThreeNeighborsAlive();
    }
}
class MoreThanThreeNeighborsAlive extends

Neighborhood {
    @Override
    public Neighborhood receiveLiveCell() {
        return this;
    }
}
```

Es necesario tener en cuenta que estos mensajes son enviados por las células al momento de formar parte del vecindario. El ida y vuelta de mensajes entre la célula y el vecindario, tal interacción, resulta en que el vecindario agregue o no a una célula viva a este.



MÉTODOS POR DEFECTO

Generalmente, cuando estamos creando una clase abstracta, es mejor definirle un comportamiento por defecto a los métodos en vez de marcarlos como abstractos. De esta manera forzamos menos a las subclases a implementar métodos que no les interesan. Así logramos que se concentren más en la funcionalidad específica y reducimos la posibilidad de código duplicado.

```
// Neighborhood implementa
public Neighborhood welcoming(Cell neighbor) {
    return neighbor.acceptedBy(this);
}

// Cell implementa
Neighborhood acceptedBy(Neighborhood neighborhood) {
    return neighborhood;
}
// LiveCell sobrescribe con
@Override
Neighborhood acceptedBy(Neighborhood neighborhood) {
    return neighborhood.receiveLiveCell();
}
```

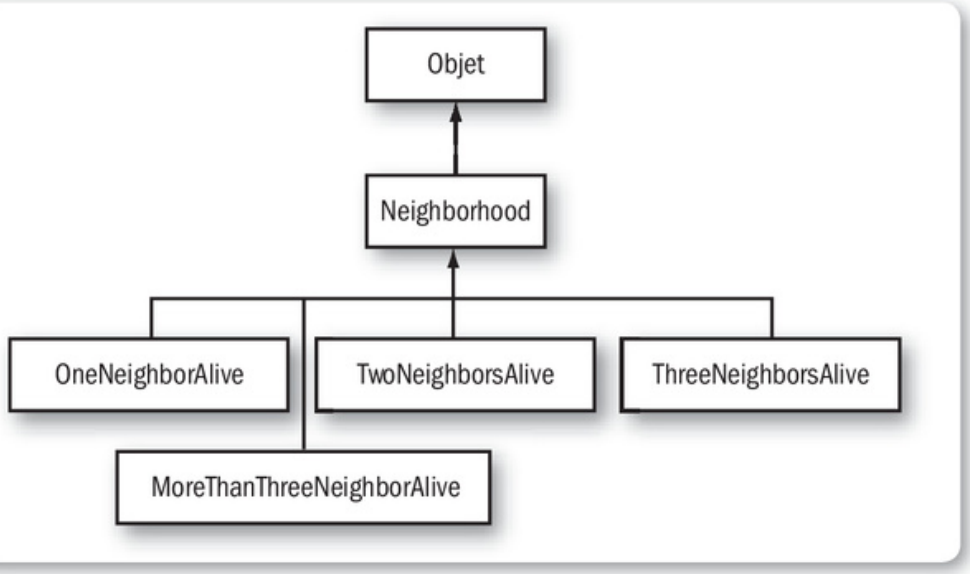


Figura 9. Esta figura nos muestra la jerarquía de las clases que representan los distintos tipos de vecindarios.

Así, la red de colaboraciones entre el vecindario y las células cuenta los vecinos vivos que tiene una célula determinada. Hasta el momento, tener todas estas clases no parece ser una gran ventaja, ya que para contar existen los números. Son útiles cuando las células colaboran con

los vecindarios para saber cómo se da la evolución. Recordemos que las células ya tiene, por su definición, el conocimiento de si están vivas o muertas, necesitamos que se lo comuniquen al vecindario. Cada vecindario responderá de la forma apropiada (según las reglas). Por lo tanto declaramos los siguientes métodos en la clase **Neighborhood**.

```
public Cell evolveLiveCell() {
    return Cell.dead();
}
public Cell evolveDeadCell() {
    return Cell.dead();
}
```

Tengamos en cuenta que no los marcamos como abstractos, de esta forma podemos darles un comportamiento por defecto y dejar que cada subclase sobrescriba lo que desee.

La implementación de las reglas ahora es trivial y directa. Cada clase de vecindario decide qué hacer en cada caso sin dudarlo.

```
class TwoNeighborsAlive extends Neighborhood {
    @Override
    public Cell evolveLiveCell() {
        return Cell.alive();
    }
}
class ThreeNeighborsAlive extends Neighborhood {
    @Override
    public Cell evolveLiveCell() {
```



VALIDAR CREACIÓN

Siempre que creamos un objeto tenemos que asegurarnos de que quede en un estado válido (para el modelo) y usable. Esto lo hacemos verificando que los argumentos de creación sean correctos, lanzando excepciones en caso contrario. De esta forma se descubren rápidamente errores en el lugar donde se inician.

```
        return Cell.alive();
    }
    @Override
    public Cell evolveDeadCell() {
        return Cell.alive();
    }
}
class MoreThanThreeNeighborsAlive extends Neighborhood {
    @Override
    public Cell evolveLiveCell() {
        return Cell.dead();
    }
}
class TwoNeighborsAlive extends Neighborhood {
    @Override
    public Cell evolveLiveCell() {
        return Cell.alive();
    }
}
class ThreeNeighborsAlive extends Neighborhood {
    @Override
    public Cell evolveLiveCell() {
        return Cell.alive();
    }

    @Override
    public Cell evolveDeadCell() {
        return Cell.alive();
```



IMPLEMENTAR TOSTRING

Es muy importante saber que todos los objetos heredan de **Object** el método **toString** que les sirva para obtener en texto una representación del objeto. En este sentido debemos tener en cuenta que este método es muy útil cuando estamos realizando el proceso de **debug**, ya que se encarga de darnos rápidamente la información importante sobre el objeto en cuestión.

```
    }
}
class MoreThanThreeNeighborsAlive extends Neighborhood {
    @Override
    public Cell evolveLiveCell() {
        return Cell.dead();
    }
}
```

Finalmente, lo que nos resta es que las células envíen el mensaje correspondiente al vecindario para evolucionar.

```
@Override // implementación en LiveCell
Cell evolveAccordingTo(Neighborhood neighborhood) {
    return neighborhood.evolveLiveCell();
}
@Override // implementación en DeadCell
Cell evolveAccordingTo(Neighborhood neighborhood) {
    return neighborhood.evolveDeadCell();
}
```

No tuvimos que realizar ningún esfuerzo para codificar el comportamiento evolutivo del juego una vez que visualizamos claramente cómo tenían que colaborar los objetos.

Ya con el código completo podemos correr nuestro test. Si queremos ver la evolución correspondiente al juego, podemos correrlo en modo **debug** y ver la variable asociada al juego en el visualizador. Si implementamos el método **toString** (en todos los objetos), veremos la grilla y las células de forma rápida y fácil.



REDUCIR EL USO DIRECTO DE CLASES

Si queremos ser flexibles, necesitamos retrasar los más posible la toma de decisiones (recordamos el late binding). Si encapsulamos la creación de objetos en métodos separados, tendremos contenida, en un solo punto, la dependencia, y por lo tanto restricción, con una clase.

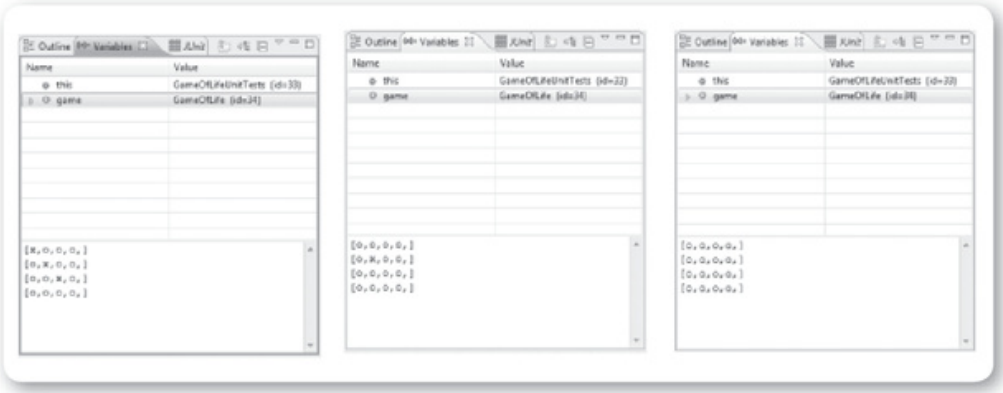


Figura 10. Implementar el método **`toString`** es muy útil para poder tener una experiencia de **debug** agradable. En esta imagen podemos darnos cuenta de la evolución de un juego.

Algunos programadores creerán que el ejercicio anterior exagera el uso de las clases y abogarán por un diseño donde no se diferencien por tipo los vecindarios y las células. Seguramente alegarán que, por un tema de performance, conviene utilizar **int** para contar los vecinos vivos y para saber si una célula está viva o muerta. En principio lo más importante de todo diseño es que modele correctamente el problema o dominio, y que haya un isomorfismo entre los elementos de uno y otro. Claramente usar números para representar vecinos no parece cumplir con esta premisa. Además, usar números para contar la cantidad de vecinos requiere que, durante la ejecución, el código se pregunte constantemente cuántos vecinos hay o si la célula está viva o muerta. Con el diseño de clases que hicimos, no necesitamos preguntarnos nada, esas preguntas ya las hicimos y las contestamos en tiempo de diseño. En la ejecución no perdemos el tiempo preguntándonos algo que ya sabemos.

> IMPLEMENTAR TOSTRING ES ÚTIL PARA TENER UNA EXPERIENCIA DE DEBUG AGRADABLE

Junto al código fuente del ejercicio pueden encontrar una implementación del juego usando **int**, así como una comparación en el tiempo de ejecución respecto de la implementación nuestra. En mi experiencia, ambas implementaciones tardan lo mismo en terminar un juego. Esto es una demostración de la falsedad del argumento de la

performace. Obtenemos los mismos resultados realizando un modelo más elaborado, entendible y extensible. Veamos cómo se ven algunos de los fragmentos de esta implementación.

```
@Override // en IntCell, que hereda de Cell
Cell evolveAccordingTo(Neighborhood neighborhood) {
    if(isDead()) {
        return neighborhood.evolveDeadCell();
    } else {
        return neighborhood.evolveLiveCell();
    }
}

static class IntNeighborhood extends Neighborhood {
    private int aliveNeighbors = 0;

   @Override
   public Cell evolveLiveCell() {
        IntCell cell = new IntCell();
        if( aliveNeighbors == 2 ||
        aliveNeighbors == 3) {
        cell.live();
    }
    return cell;
}
@Override
public Cell evolveDeadCell() {
    IntCell cell = new IntCell();
```



RUNTIME VERSUS DISEÑO

Generalmente los **if** presentan una oportunidad para mejorar el diseño y evitar preguntar una y otra vez lo que ya se sabe. Este esfuerzo por construir un contexto de conocimiento es una pérdida de recursos (procesamiento). Es mejor tratar de llevar esas decisiones al plano del diseño.

```
        if(aliveNeighbors == 3) {
            cell.live();
        }
        return cell;
    }

    @Override
    public Neighborhood receiveLiveCell() {
            aliveNeighbors++;
            return this;
       }
    }
```

Tanto la nueva célula como el vecindario fueron implementados como **protected static classes**, para que estuvieran contenidas en la clase juego y no pudieran ser accedidas desde afuera.

Analicemos ahora nuestra implementación del juego de la vida basándonos en ciertas propiedades deseables que deberían cumplir los buenos sistemas y las partes que lo componen.

1) Descomposición
2) Composición
3) Entendimiento
4) Continuidad
5) Isomorfismo
6) Bajo acoplamiento
7) Alta cohesión
8) Única responsabilidad
9) Principio Abierto/Cerrado



CLASES DE DATOS

Si una clase solo tiene getters y setters, se la conoce como clase de datos. Estas clases generalmente no tienen más responsabilidad que contener datos, estando la lógica que opera sobre estos en otro lado. Estas clases tienen que ser refactorizadas y mover esa lógica a ellas.

1) Un sistema tiene que poder ser descompuesto en varias partes, menores y de menor complejidad, relacionadas entre sí. Estas partes tienen que ser lo más independientes posible unas de las otras.

En nuestro caso, el cliente del juego lo ve como una sola entidad, la clase **GameOfLife**, pero en realidad el juego está implementado por otros varios elementos (la grilla, las filas, las células y los vecindarios). Y hay una gran independencia de la grilla y las filas respecto de las células y los vecindarios. Además las células y los vecindarios solo se conocen entre ellos y desconocen al juego en sí, a la grilla, filas y demás elementos.

2) Se busca que los elementos complejos sean creados por la combinación libre de elementos menos complejos, con otros distintos.

La implementación del juego propuesta está basada en la combinación de células y vecindarios distintos, cada uno con un propósito diferente. Y también en agregar la grilla, compuesta de filas, que sólo tiene conocimiento de las células por el hecho de ser su contenedor en este caso particular.

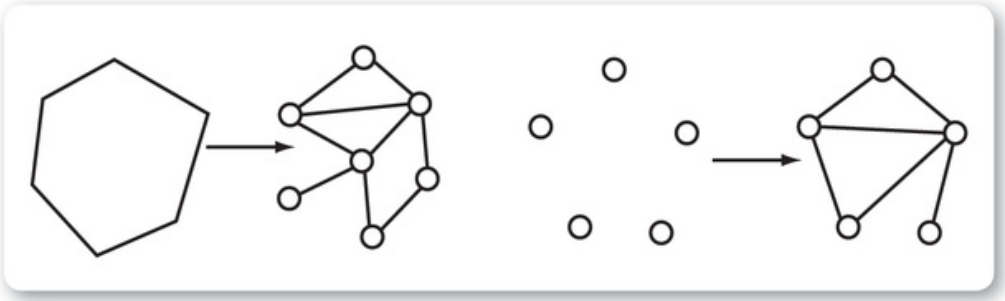


Figura 11. `Descomposición` y **`Composición`**, dos conceptos distintos que pueden llegar a confundirse.



PROTOCOLOS REDUCIDOS

Es muy importante tener en cuenta que si el bajo acoplamiento dice que debemos reducir la cantidad de relaciones que tiene una clase con otras, también es conveniente que estas dependencias sean lo más abstractas posible. Cuanto más abstracta es una relación, menos métodos debería tener, debería describir un protocolo más chico. Esto también ayuda a la hora de hacer cambios.

3) Una persona tiene que poder entender cada elemento que conforma un sistema, de forma independiente y separada del resto de los componentes. Unos poco elementos extras son tolerables. No tendría que ser necesario estudiar todo el sistema para poder comprender la funcionalidad de una de sus partes.

Recordemos que la grilla, las filas y también las posiciones, son fácilmente compresibles sin necesidad de ver otras clases. En el caso de las células y los vecindarios, al estar tan íntimamente relacionados, es obligatorio analizarlos en conjunto, aunque solamente se debe analizar a las clases base **Cell** y **Neighborhood**.

4) Continuidad se refiere a que si hubiera un pequeño cambio en los requerimientos de un sistema, debería repercutir en él como un pequeño cambio también. Si tenemos un cambio que en el dominio es pequeño y en nuestro sistema repercute en muchos lados, es claro que nuestro diseño no lo ha modelado correctamente.

No solo para cambios, sino también para nuevos descubrimientos sobre el dominio deben contemplar la continuidad.

En nuestro diseño es fácil observar que un cambio en las reglas es fácilmente aceptado, ya que tenemos bien discriminados los vecindarios y las células. Podríamos agregar más estados en las células, otros vecindarios, cambiar la forma de la grilla de cuadrada a hexagonal, etcétera.

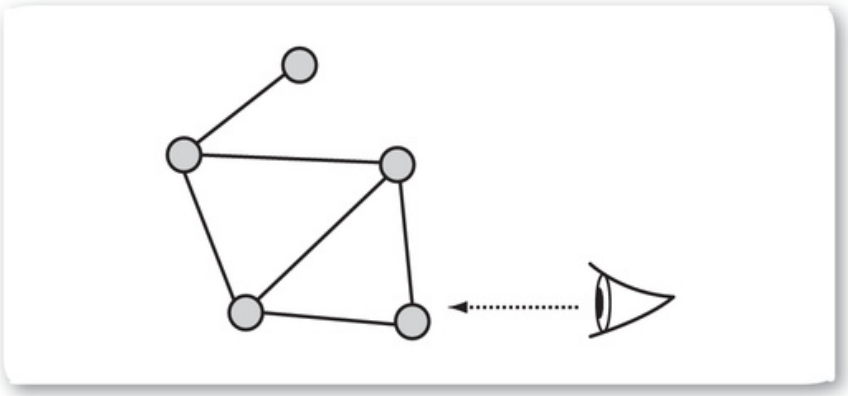


Figura 12. Para entender un parte del sistema solamente se debería ver esa parte y a lo sumo algunos otros componentes vecinos. Nunca todo el sistema.

5) Ya hemos hablado de este tema. Debemos construir una correlación entre los elementos del dominio y los elementos de nuestro modelo.

Nuestro diseño respeta los elementos que conforman el juego, así como sus reglas. Todos ellos están representados en nuestro modelo.

6) Bajo acoplamiento significa que los elementos deben tener poca dependencia entre sí. Cuanta más dependencia tiene un elemento, más restricciones tiene al cambio. Asimismo los elementos de los que depende también están restringidos a cambiar libremente. A más dependencia, obtendremos menos flexibilidad.

Por nuestra parte tenemos que **GameOfLife** tiene conocimiento de las células, los vecindarios y la grilla, los elementos que son importantes en el juego. Desconoce las filas, por ejemplo. Las células sólo conocen al vecindario y viceversa.

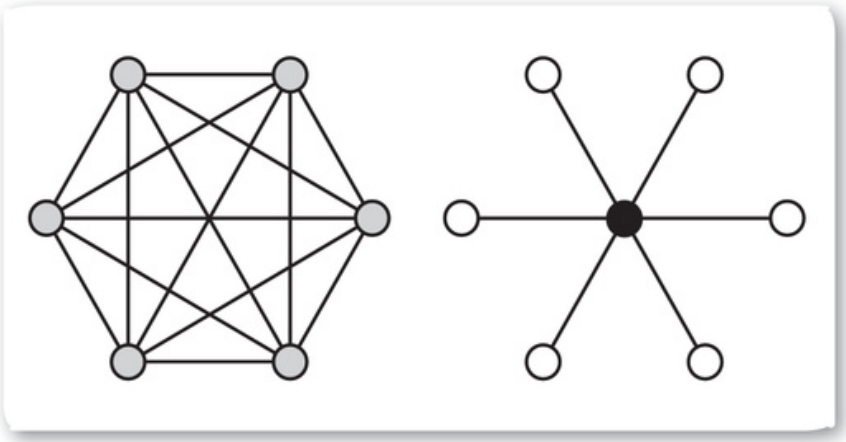


Figura 13. Cuantas más conexiones existen entre los elementos, más rígidos se vuelven frente a los cambios.

7) Cuando hablamos de alta cohesión, estamos hablando de que las funcionalidades y capacidades similares y con el mismo propósito deben estar lo más cerca posible. Esto significa que los elementos que están relacionados deben, en conjunto, cumplir una tarea específica. Y su existencia tiene que estar para cumplir con esa funcionalidad.

Un ejemplo claro de esto es que las células solamente conocen al vecindario, y viceversa. Y solamente lo conocen para poder establecer el siguiente paso en la evolución de la célula.

8) Una clase debe tener una única responsabilidad. Si una clase sirve para varias cosas (distintas) está teniendo un problema de múltiples personalidades. Al tener una única responsabilidad, es más fácil entender el objetivo de esta y es más difícil cometer errores con ella.

Cada clase de nuestro diseño responde a una única responsabilidad.

9) El principio **Abierto/Cerrado** dice que un elemento debe estar abierto para la extensión y cerrado para su modificación. El elemento tiene que ser una caja negra para los usuarios de este. Debe ser posible extender sus funcionalidades, pero no realizar modificaciones internas.

Al tener cada estado de una célula separado en célula viva y en célula muerta, y los vecindarios correctamente enumerados, es posible crear nuevas clases que los modelen.



RESUMEN

Este capítulo nos acompañó durante el proceso para comprender cómo definir clases abstractas como también clases anidadas y clases anónimas. Estas son poderosas herramientas ya que nos permiten encapsular mejor el conocimiento y tenerlo cerca de donde se utiliza. El ejercicio que presenta el capítulo es de gran importancia ya que revela las ventajas de utilizar un buen modelo y de aplicarlo con las técnicas adecuadas. Nos muestra que modelar correctamente el dominio del problema, creando todas las clases que sean necesarias para corresponder a un ente del dominio, tiene grandes beneficios sin impactar negativamente en la performance de nuestro código.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Qué es una clase abstracta?
2	¿En qué se diferencia una clase estática de una interna?
3	¿Para qué se utilizaría una clase anónima?
4	¿Debemos evitar los ifs? ¿Cuándo tiene sentido usarlos?
5	¿Tiene sentido marcar una clase como abstract y final al mismo tiempo?
6	Enumere algunas ventajas de usar **static factory methods**.
7	¿Cuál es el problema de que un objeto tenga muchos atributos?
8	Si un cambio en una clase repercute en muchas otras, ¿qué está pasando?
9	¿A qué nos referimos cuando hablamos de alta cohesión y bajo acoplamiento?
10	¿Qué dice el principio **Abierto/Cerrado**?

ACTIVIDADES PRÁCTICAS

1	Implemente una grilla cuadrada alternativa a la original.
2	Cámbielo de modo que el vecindario sean las células que están arriba, abajo, a la derecha y a la izquierda solamente.
3	Implemente un método que evolucione el juego hasta que estén todas las células muertas.
4	Encárguese de implementar un mecanismo de frenado al método anterior por un número de turnos.
5	Genere una grilla hexagonal.

*06 –

Interfaces

Una de las características que Java evitó copiar de C++ fue permitir la herencia múltiple de clases. Si bien la herencia múltiple puede ser una poderosa herramienta, en la práctica presenta problemas. Uno de los más conocidos es cuando una clase hereda de jerarquías distintas que contienen métodos con la misma firma. En este capítulo enfrentaremos este problema con los mecanismos denominados interfaces.

- Definición 124
- Uso 126
- Clases abstractas versus interfaces 129
- Resumen 131
- Actividades 132

Definición

Una interfaz es solamente la declaración de los mensajes que sabe responder un determinado tipo, es decir, es la declaración del protocolo aceptado por los objetos de ese tipo. Las interfaces no suplantan a las clases, las complementan. Una clase, que es protocolo más comportamiento, implementa una o varias interfaces, que solamente son protocolo. Para implementar un interfaz, una clase tiene que poder responder los mensajes declarados en ella, ya sea implementándolos o definiéndolos como abstractos. Las interfaces, al igual que las clases abstractas no pueden tener instancias directamente. Es posible hacer que una interfaz extienda otra, requiriendo que la clase que implementa tal interfaz implemente también la que extiende.

RECORDEMOS QUE UNA CARACTERÍSTICA DESEABLE DE UN SISTEMA ES EL BAJO ACOPLAMIENTO

Como ya dijimos, las interfaces declaran la firma de los mensajes, no contienen implementación alguna. De esta forma estas no sirven para reutilizar código directamente, pero sirven para otras funciones más importantes.

Tengamos en cuenta que las interfaces sirven para reutilizar código mientras escapamos de las restricciones propias de la herencia de clases.

Recordemos que en el capítulo anterior dijimos que una característica deseable de un sistema era el bajo acoplamiento. Podemos reducir el acoplamiento disminuyendo la cantidad de clases de las cuales dependemos. También podemos bajar el nivel de acoplamiento haciendo que las dependencias sean con tipos de protocolo reducido. Los protocolos reducidos no solo alivianan las dependencias, sino que también hacen que un



INSTANCIAS DE UNA INTERFAZ

Una interfaz por sí sola no puede tener instancias, sino que depende de una clase que la implemente. Podemos probar tratando de instanciar una y ver que nos responde el compilador o Eclipse. Además pensemos, si una interfaz solo define protocolo, firmas de métodos y mensajes que debe entender, ¿qué es lo que ejecutaría una instancia de tal interfaz?

determinado tipo sea más entendible, ya que tenemos que comprender menos mensajes para saber cómo responde un objeto. Las interfaces son artefactos que permiten agrupar mensajes relacionados y separarlos de otros mensajes no relevantes para una determinada funcionalidad. Una clase que implementa varias interfaces tiene la capacidad de responder a varias funcionalidades. Pero los clientes de dicha clase no tienen por qué saber todo lo que ella puede hacer, los clientes solamente deben conocer la interfaz que les interesa y nada más. Así evitamos engrosar las dependencias entre los elementos, ya que los clientes son independientes de los cambios que puedan ocurrir en los otros mensajes.

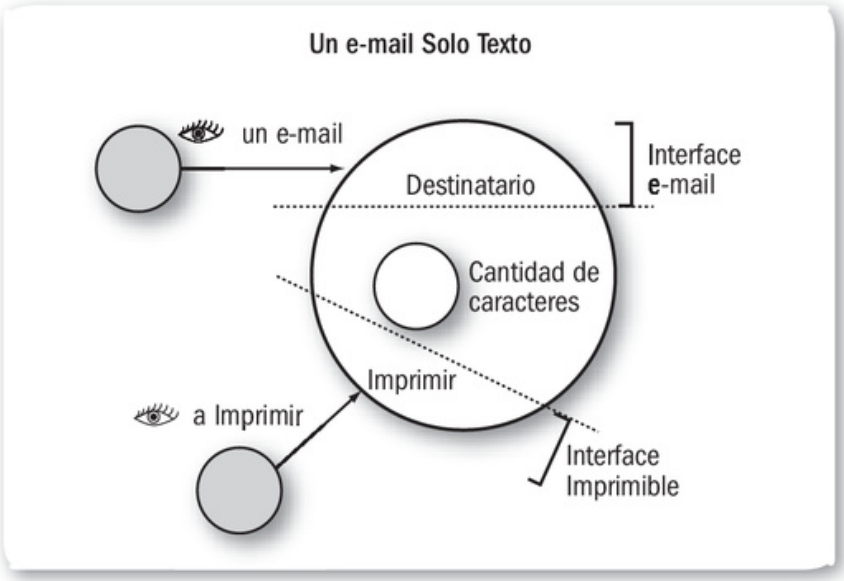


Figura1. Las interfaces permiten restringir el protocolo usado por un objeto cliente a los mensajes necesarios solamente para una funcionalidad.

Para Java, las interfaces también son tipos como los son las clases, por lo tanto los objetos pertenecen tanto al tipo determinado por su clase, así como también, a los tipos establecidos por las interfaces que implementan. Estos forman jerarquías que no siguen la línea de las clases, sino que hay una jerarquía por interfaz y por las clases que las implementan. De esta forma podemos decir que se trata de una relación de tipos que es transversal a la clasificación.

Uso

Para definir una interfaz en Java utilizamos la palabra interface en vez de **class**. Así mismo acepta los distintos modificadores de visibilidad.

```
public interface Mensaje {
    ...
}
```

Ahora la definición del protocolo se realiza con las firmas de los métodos, sin cuerpo, de igual forma que cuando definimos un método abstracto. Por definición todos los métodos indicados en una interfaz son abstractos, por lo que utilizar este modificador es redundante. Así mismo ocurre con la visibilidad de los métodos, todos son públicos.

```
public interface Mensaje {
    Destinatario destinatario();
        ...
}
```

Eclipse nos facilita la tarea de crear una interfaz desde cero. Para ello hacemos clic en el menú **File** o sobre el botón de **New** en la barra de herramientas. También podemos hacer clic con el botón secundario del mouse y seleccionamos **New.../Interface**. Elegimos la carpeta donde se encuentran los archivos fuentes del proyecto y el paquete de destino, luego ingresamos el nombre de la interfaz y seleccionamos posibles interfaces a extender. Para terminar presionamos **Finish**.



INTERFACES CON CONSTANTES

Si bien es posible definir constantes en las interfaces, no es una práctica del todo recomendable. Generalmente las interfaces que son puramente constantes pueden ser reemplazadas por enumeraciones (que veremos más adelante) o moviendo las constantes a alguna clase abstracta o a la clase que las utiliza. Además recordemos que no es posible sobrescribir una constante y que no se adaptan bien al cambio.

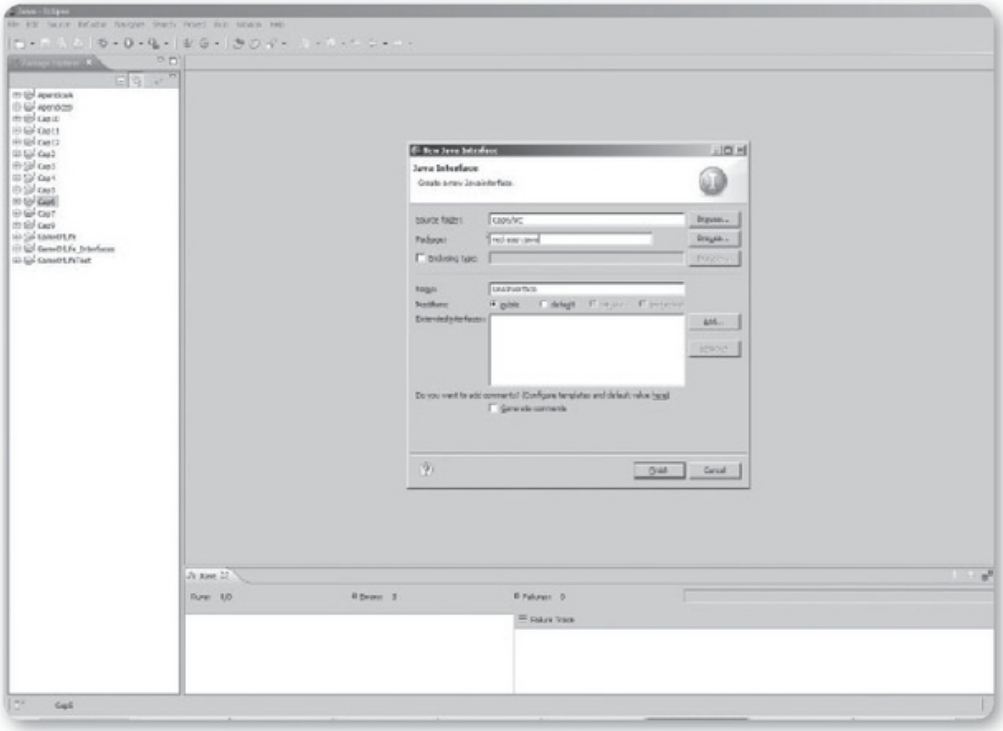


Figura 2. En esta imagen podemos ver la ventana que corresponde a la pantalla de creación de interfaces.

En las interfaces no se pueden definir atributos de instancia, solamente métodos. Tampoco se pueden definir métodos estáticos. Lo único que se puede definir en una interfaz, además del protocolo, es una constante. Las constantes son atributos públicos y estáticos, generalmente se escriben totalmente en mayúscula y se los marca como **final**.

```
public interface MiInterfaceConConstantes {
    static final double PI = 3.14d;
    ...
}
```

Igual que en el caso del protocolo, el atributo **public** es redundante ya que todo en una interfaz corresponde a público.

Como dijimos anteriormente, una interfaz no hereda de otra (o de muchas otras), sino que la **extiende**. Cuando una interfaz extiende a otra, amplia el protocolo definido por esta con nuevos métodos.

```
public interface Email extends Mensaje {
    ... // defino nuevo mensajes
}
```

Es necesario tener en cuenta que antes de que aparecieran las anotaciones en Java, uno de los usos de las interfaces era para marcar a las clases con información. Un ejemplo que está en el SDK es la interfaz **Serializable**. Estas interfaces de marca están absolutamente vacías y solo aportan por su presencia. En este sentido debemos tener en cuenta que en la actualidad lo mejor es utilizar anotaciones para este propósito, ya que se trata de una opción más poderosa.

Si bien se utiliza la misma palabra, **extends**, para extender una interfaz y para heredar de una clase, la semántica es bien distinta. En un caso extendemos un protocolo, en el otro se crea una jerarquía de clases, sin necesidad de extender protocolo.

Cuando una clase quiere implementar una o varias interfaces lo hace mediante la palabra implements, seguida de los nombres de las interfaces que desea implementar separados por coma.

```
public class EmailSoloTexto implements Email {
    @Override // implemento los mensajes de la interfaz
    public Destinatario destinatario() {
    ...
    }
    ...
}
```

Al igual que cuando implementamos un método abstracto o sobrescribimos un método de alguna clase padre, es conveniente anotar el método que implementa un mensaje de una interfaz con **@Override**. También debemos acordarnos de agregar el modificador de visibilidad **public**, de esta forma no romperemos el contrato con la interfaz, ya que es necesario que sea público.

Es posible utilizar una interfaz para definir una clase anónima en un método. El mecanismo es el mismo que vimos anteriormente.

```
...
Mensaje mensaje = new Mensaje() {
        @Override
        public Destinatario destinatario() {
        ...
        }
...
};
...
```

Clases abstractas versus interfaces

Es difícil, al principio, determinar cuándo es necesario, o conveniente, crear una interfaz o crear una clase abstracta. Primero recordemos que ambos artefactos tienen propósitos distintos. Las clases abstractas sirven para establecer una jerarquía de tipos y proveer un comportamiento básico. Las interfaces son definiciones de protocolos, determinan el comportamiento de un tipo. Segundo, las clases abstractas pueden incluir código, definir atributos, tener métodos estáticos, etc. En cambio, las interfaces son sólo declaraciones de mensajes.

Por este motivo las interfaces son mucho menos restrictivas que las clases abstractas, ya que no fuerzan nada más que lo necesario, solamente el protocolo. Por ejemplo, una clase abstracta podría definir un atributo de instancia, eso ya es una restricción de implementación,



FACILIDAD PARA TESTEAR

Es importante tener en cuenta que una de las grandes ventajas de usar interfaces en vez de referenciar directamente clases es que a la hora de escribir nuestros test, estas dependencias son mucho más fáciles de proveer. Solamente es necesario crear un simple objeto que implemente la interfaz, cosa que podemos hacer incluso en el mismo método con clases anónimas.

ya que todas las subclases tienen como peso ese atributo (incluso si no lo usan). Además, la herencia es una clase que se puede utilizar una sola vez. Solo podemos heredar de una clase. Podemos implementar todas las interfaces necesarias. Nuestra clase, es entonces, polifacética. Ya que puede servir a muchos más clientes. Generalmente, si nuestra clase abstracta solamente define métodos, lo mejor es crear una interfaz y reemplazarla. Así ganamos un poco de flexibilidad y dejamos la clase abstracta para cuando queremos reutilizar comportamiento (código) en común entre varias clases similares.

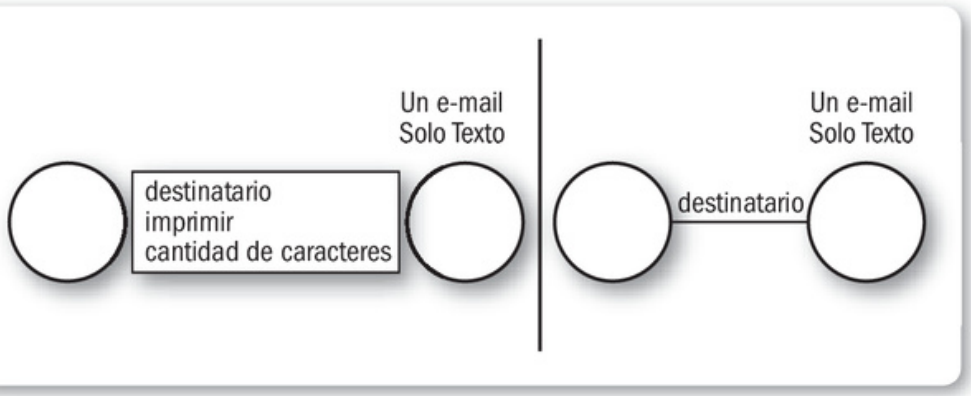


Figura 3. Debemos tener en cuenta que las interfaces se encargan de reducir el acoplamiento entre dos objetos.

Una forma de saber cuándo debemos crear una interfaz es cuando estamos programando una clase y sabemos que vamos a necesitar ciertos colaboradores que todavía no existen. En ese caso es cuando podemos crear una interfaz. Así nuestra clase compilará habiendo ya creado los artefactos mínimos para eso.

NIVEL DE ABSTRACCIÓN

Cuando diseñamos interfaces, al igual que cuando lo hacemos con las clases, debemos ser los más abstractos posible. Eso significa que en una interfaz debemos definir el mínimo conjunto de mensajes que tengan sentido, que estén relacionados bajo un concepto, y evitar agregar métodos extraños. En esos casos es mejor extender la interfaz y agregarlos en la nueva.

Si queremos definir una interfaz a partir de una clase existente y que esta la implemente, Eclipse tiene una funcionalidad para este propósito.

Para extraer una interfaz debemos presionar el botón derecho del mouse sobre el código de una clase y seleccionamos la opción **Refactor/ Extract Interface**. Luego ingresamos un nombre para la interfaz y elegimos qué métodos de la clase pertenecerán a esta, hacemos clic sobre **OK**.

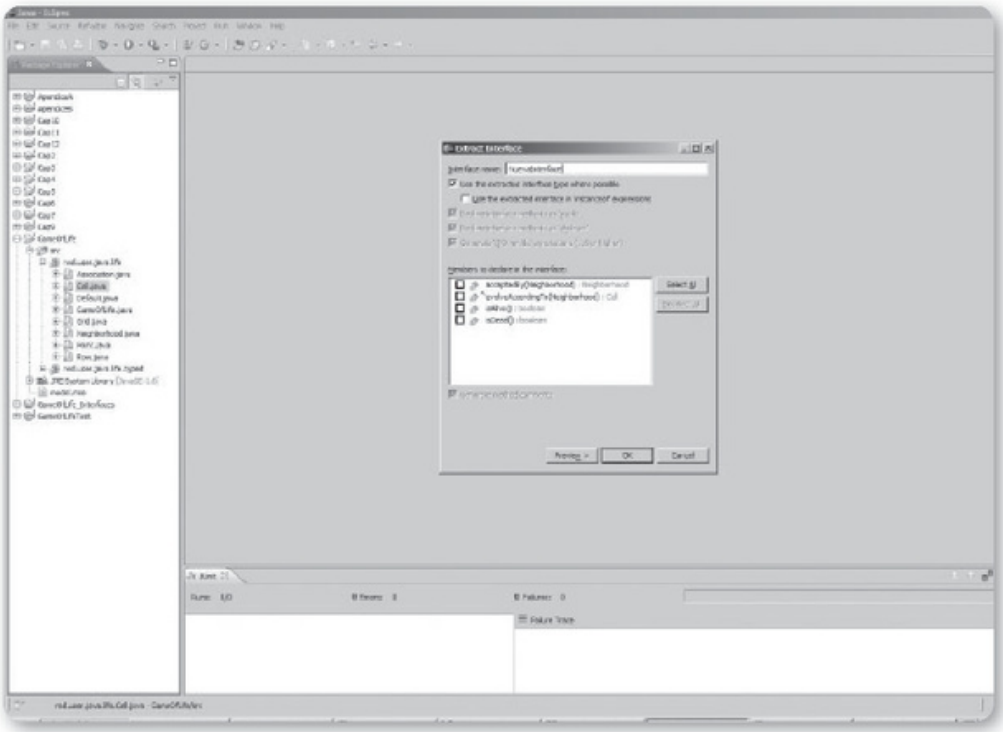


Figura 4. Aquí podemos ver la pantalla que nos permite realizar la extracción de interfaz a partir de una clase.

RESUMEN

Las interfaces son un poderoso mecanismo de polimorfismo para un lenguaje estáticamente tipado como Java, ya que permiten salirse de la estructura de las jerarquías de clases. Son la forma de tipado más simple y poderosa de Java porque solamente definen los mensajes que conforman el protocolo sin imponer más restricciones a los objetos que las componen. Al poder una clase implementar muchas interfaces resulta más flexible que la herencia. Aunque las interfaces también tienen limitaciones frente a las clases, ya que no pueden contener código ni atributos, entre otras cosas.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Qué es una interfaz?
2	¿Qué elementos puede contener un interfaz?
3	¿Puedo declarar métodos privados en una interfaz?
4	¿Cuántas interfaces puede implementar una clase?
5	¿De cuántas interfaces se puede hacer la extensión de otra?
6	¿Pueden tener métodos con código las interfaces?
7	Enumere algunas diferencias entre las interfaces y las clases abstractas.
8	¿El primer paso al definir un concepto sería crearlo como interface?
9	¿Cuándo es el momento de crear una interface?
10	¿Cómo ayudan las interfaces a reducir el acoplamiento?

ACTIVIDADES PRÁCTICAS

1	Utilice el ejercicio de El juego de la vida y cambie la clase Default a una interfaz.
2	Mueva el método toNull a una clase nueva.
3	Remueva el método toNull y cree una clase propia para ese Default.
4	Cambie la jerarquía de Neighborhood para que sea una interfaz, una clase abstracta y el resto de las clases existentes.
5	¿Podría Cell ser una interfaz? Haga la prueba.

*07

Enumeraciones

Las enumeraciones son la solución Java, desde su versión 5, al viejo problema de utilizar números enteros o Strings para representar un conjunto de elementos acotado y bien definido donde cada uno tiene un nombre. Son seguras de usar ya que tienen un tipo explícitamente definido para ellas, no pudiéndose intercambiar con otros tipos y cometer errores.



- Definición 134
- Uso 136
- Resumen 147
- Actividades 148

Definición

Es común que ciertos dominios que se nos presentan a la hora de modelar tengan elementos distinguidos. Imaginemos una baraja de cartas. Los naipes en cuatro clases: corazón, trébol, pica y diamante, y además numerados del dos al nueve, seguido de las figuras J (sota), Q (reina), K (rey) y A (as). Aquí tenemos dos ejemplos de enumeraciones, los cuatro palos y las diez cartas. El palo trébol es un elemento distinguido del conjunto de palos, del mismo modo que el número cinco, lo es del conjunto de la numeración de las cartas. Las enumeraciones permiten entonces definir un conjunto de elementos distinguidos. En las versiones anteriores de Java, este tipo de enumeraciones se definían utilizando constantes de tipo numérico o de tipo **String**.

```
public class Palo {
    public static final String PICA = "♠";
    public static final String CORAZON = "♥";
    public static final String DIAMANTE = "♦";
    public static final String TREBOL = "♣";
}
```

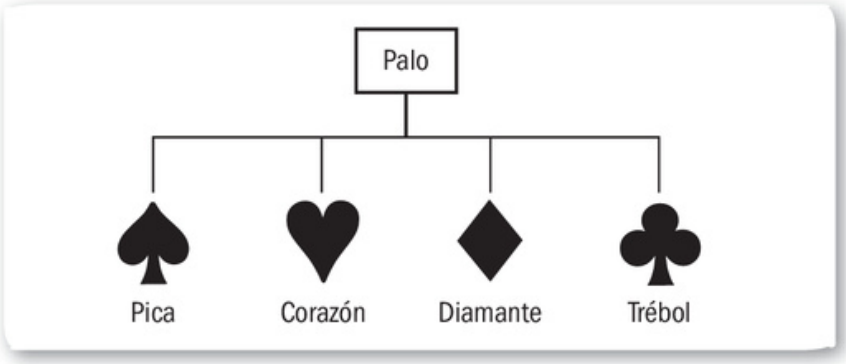


Figura 1. Esta imagen nos muestra el diagrama que corresponde a una enumeración con el estilo viejo.

El problema de este tipo de enumeraciones es que **PICA** no es de tipo **Palo** y por lo tanto en todos lados donde queramos usar un palo aparecerá el tipo **String**. Imaginen un método que está esperando un

tipo Palo y se encarga de recibir la cadena denominada **"hola mundo!"**, seguramente se encargaría de romper el programa.

Es este sentido, sabemos que una solución para estos casos era utilizar realmente el tipo **Palo** y proceder a hacer que cada palo sea una instancia. Para asegurarse de que no se crean más palos que los determinados, se marca como privado el constructor.

```
public class Palo {
public static final Palo PICA = new Palo("♠");
        public static final Palo CORAZON = new Palo("♥");
        public static final Palo DIAMANTE = new Palo("♦");
        public static final Palo TREBOL = new Palo("♣");

private String símbolo;

private Palo(String símbolo) {...}
...
}
```

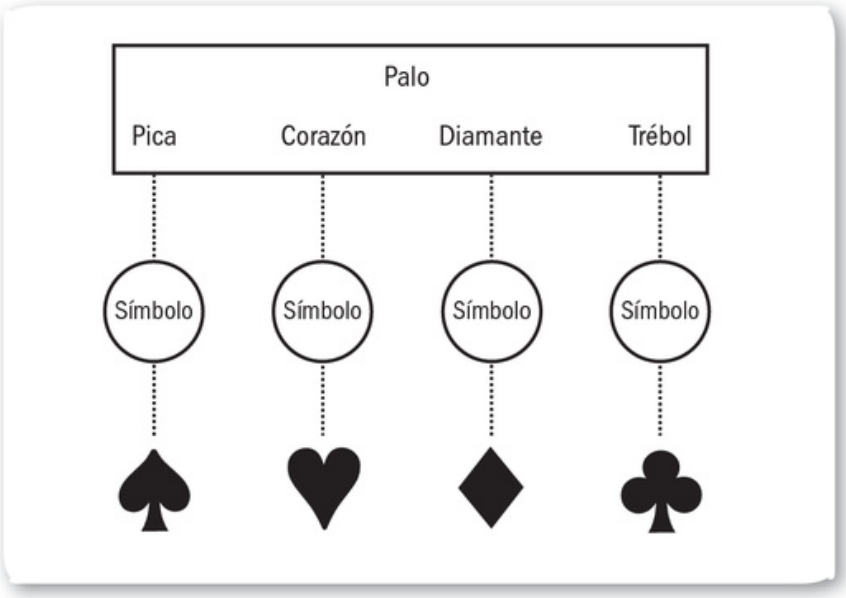


Figura 2. Diagrama de una enumeración utilizando objetos que modelan correctamente el concepto.

Este tipo de construcciones representan una gran mejora frente a las otras enumeraciones, ya que tenemos un tipo determinado para representar a los palos e instancias de ese tipo. Las enumeraciones modernas en Java son un artefacto sintáctico que produce el mismo resultado que la construcción anterior pero sin escribir tanto. El lenguaje ahora tiene el concepto de enumeración, no es algo que solamente esté en la mente del programador.



Uso

A partir de la versión 1.5, definir una enumeración en Java se consigue mediante el uso de la palabra clave **enum** en lugar de **class**. Luego en el cuerpo de la enumeración se definen los elementos distinguidos.

```
public enum DiaDeLaSemana {
   DOMINGO, LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO
}
```

Como vemos, cada día de la semana así definido se presenta como un objeto instancia que corresponde a **DiaDeLaSemana**.

Eclipse facilita la creación de enumeraciones al proveer una ventana para esta tarea. Para crear una enumeración en Eclipse, vamos al menú File o al botón New en la barra de herramientas. Elegimos la carpeta donde se encuentran los archivos fuentes del proyecto y seleccionamos el paquete destino. Luego escribimos el nombre de la enumeración y elegimos las posibles interfaces que implementará la enumeración. Para terminar presionamos el botón **Finish**.



CONVENCIÓN

Dado que generalmente las enumeraciones representan entes constantes e inmutables, es convención aceptada que los nombres de los elementos estén totalmente escritos en mayúsculas. Esto facilita, en la lectura, reconocer qué elementos son enumeraciones y cuáles no, sin tener que navegar todo el código.

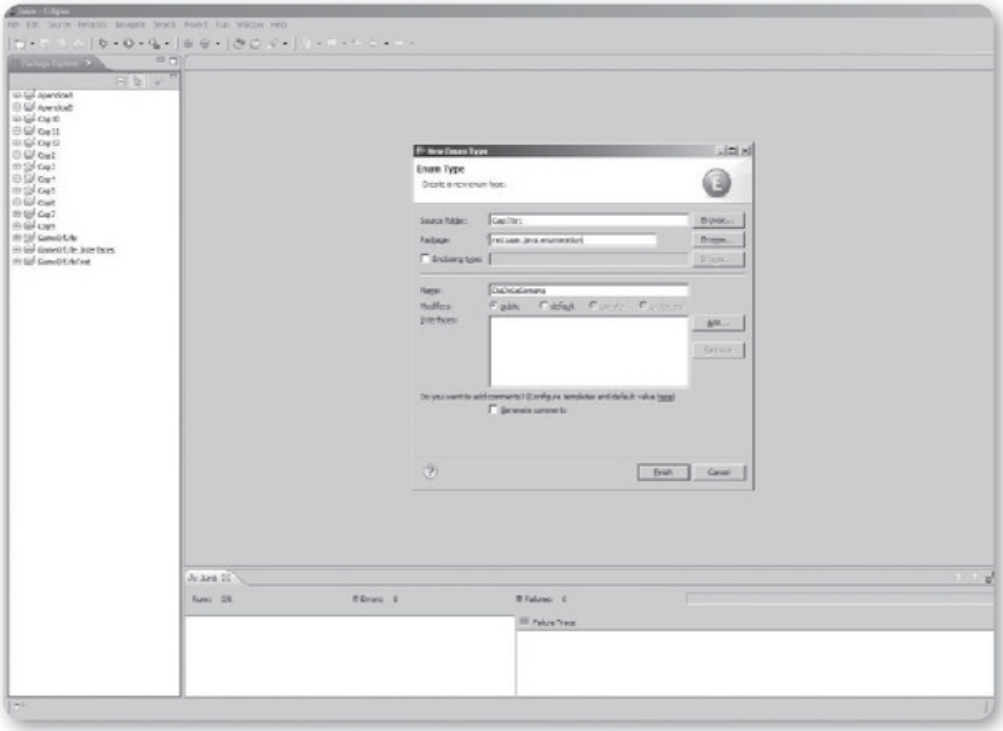


Figura 3. Esta imagen nos muestra la vista de la ventana de creación de enumeraciones en el entorno de desarrollo **Eclipse**.

Al ser, también clases, las enumeraciones pueden definir sus propios métodos y atributos, así como los constructores.

```
public enum Palo {
    PICA("♠"),
    CORAZON("♥"),
    DIAMANTE("♦"),
    TREBOL("♣");
    private String simbolo;
    Palo(String simbolo) {
        this.simbolo = simbolo;
    }
    public String simbolo() {
        return simbolo;
    }
    @Override
```

```
    public String toString() {
    return simbolo();
    }
}
```

Los constructores en las enumeraciones pueden solamente ser privados o tener privacidad default. No es posible hacerlos públicos porque entonces con ellos se podrían crear otros elementos. Como se ve en el ejemplo, las enumeraciones también son clases. En consecuencia es posible hacer que implementen interfaces.

```
public interface Palo {
    String simbolo();
}
```

Debemos tener en cuenta que si queremos implementarla para los palos de la baraja inglesa será necesario proceder como sigue:

```
public enum PaloIngles implements Palo {
    PICA("♠"),
    CORAZON("♥"),
    DIAMANTE("♦"),
    TREBOL("♣");

    ...
}
```



UTILIZAR IMPORT STATIC

Es necesario recordar que cuando nos encargamos de trabajar con elementos distinguidos es conveniente que importemos estáticamente la enumeración a la cual pertenecen. De esta forma la escritura del código que los utiliza es mucho más sencilla, así como también su lectura, ya que no hay que estar leyendo una y otra vez el nombre de la enumeración como prefijo.

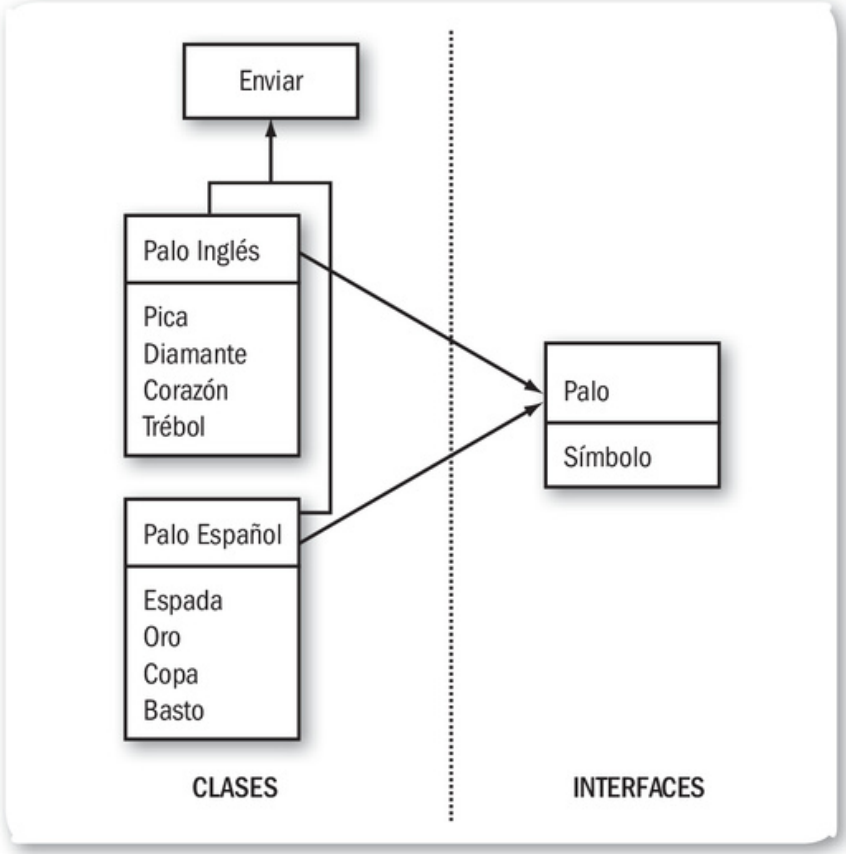


Figura 4. Es posible implementar interfaces en las enumeraciones para formar relaciones de tipo.

Las enumeraciones heredan automáticamente de la clase **Enum** de Java y no se les puede incorporar ninguna extensión, ni siquiera de otra enumeración, ya que son clases finales.

Por heredar de **Enum**, los elementos de una enumeración tienen varios métodos interesantes. El primer método es **name()**, el cual devuelve el nombre con el que identificamos el elemento.

ACLARAR ARGUMENTOS: ENUMERACIONES

En muchas **API** y código que podamos escribir existen métodos que, dado un parámetro, como **boolean**, hacen una cosa u otra. El cliente de ese código tiene que saber qué significan esos parámetros para usar los métodos. En esos casos es mejor utilizar una enumeración que refleje la intención de las opciones.

```
// true
assertEquals(Palo.PICA.name(), "PICA");
```

El otro método es **ordinal()**, que permite conocer el orden en el cual fue definido el elemento, comenzando por el cero.

```
// true
assertEquals(Palo.DIAMANTE.ordinal(), 2);
```

La clase **Enum** también provee algunos métodos estáticos útiles para manejar las enumeraciones. Uno es el método **values()**, que retorna un **array** con todos los elementos de la enumeración, en el orden que fueron definidos. El segundo es **valueOf(String)** que permite obtener el elemento que se llama como el **String** entregado como argumento. Si el nombre entregado no coincide, arroja una **IllegalArgumentException**.

```
// true
assertArrayEquals(Palo.values(), new Palo[] { PICA, CORAZON, DIAMANTE,
    TREBOL });
// true
assertEquals(TREBOL, Palo.valueOf("TREBOL"));
// arroja una IllegalArgumentException
Palo.valueOf("trebol");
```

Los elementos de una enumeración pueden están ordenados dado el orden de definición y por lo tanto se los puede comparar con el método **compareTo(Enum)** de la interfaz **Comparable** que **Enum** implementa.



ENUMERACIONES SIN COMPORTAMIENTO

Algunas enumeraciones, como los días de la semana, no tienen ningún comportamiento aparente. Solo existen por el hecho de asignar un día a un objeto. Este tipo de enumeraciones lleva a código con **if** y **switch**s en vez de usar el polimorfismo al asignarle métodos con comportamiento distinto a los elementos.

```
public enum DEFCON {
    FADE_OUT(5),
    DOUBLE_TAKE(4),
    ROUND_HOUSE(3),
    FAST_PACE(2),
    COCKED_PISTOL(1);

    ...
}
```

En el ejemplo se observa una enumeración para los niveles de **DEFCON** (niveles de defensa de los Estados Unidos), desde el nivel **DEFCON 5** que es el estadio normal, hasta **DEFCON 1** que significa guerra. **DEFCON 5** es por lo tanto el menor nivel de alerta, lo cual se re□eja en su orden de definición.

```
// true
assertTrue(FADE_OUT.compareTo(COCKED_PISTOL) < 0);
```

Evitar la utilización del orden de las definiciones como información adicional, como muestra el siguiente ejemplo, ya que puede cambiar y afectar código escrito que depende de eso.

```
public enum Piso {
    PB, PRIMERO, SEGUNDO, TERCERO;
    public int numero() { return ordinal(); }
}
```



IGUALDAD E IDENTIDAD

Los elementos de una enumeración son entidades únicas y, por lo tanto, siempre que usemos alguno de ellos estaremos usando la misma instancia. Es así que utilizar la igualdad (método equals) o utilizar la identidad (operador ==) es indistinto, ya que arrojan los mismos resultados.

Debemos tener en cuenta que lo recomendable es utilizar atributos para guardar este tipo de información.

```
public enum Piso {
    PB(0), PRIMERO(1), SEGUNDO(2), TERCERO(3);

    private int numero;

    Piso(int numero) { this.numero = numero; }

    public int numero() { return numero; }

}
```

Las enumeraciones son soportadas por la estructura switch de forma natural. Aunque esta no es la mejor forma de realizar una determinada acción, en base al elemento de la enumeración.

```
enum Operación {SUMA, RESTA, MULTIPLICACION, DIVISION}
...
switch(unaOperacion) {
    case SUMA: a + b; break;
    case RESTA: a – b; break;
    case MULTIPLICAION: a * b; break;
    case DIVISION: a / b; break;
    default: throw new AssertionError("Operación desconocida");
}
```



CREAR MÉTODOS DE BÚSQUEDA

Recordemos que cuando nuestros elementos distinguidos tienen atributos asociados a ellos, es conveniente proveer uno o varios métodos que, dado alguno de los atributos, devuelva el elemento correspondiente. De forma similar a la que trabaja **valueOf**, que dado el nombre devuelve el elemento. De esta forma podemos sobrecargar **valueOf** siempre que sea posible.

En este sentido, si fuéramos a agregar un nuevo elemento a la enumeración sería necesario proceder a buscar en todos los lugares donde usamos el **switch** y agregarlo ahí.

Debemos tener en cuenta que una gran característica de estas enumeraciones es que podemos realizar la especificación del comportamiento por instancia, como si estuviéramos utilizando clases anónimas para definirlas. De esta forma aprovechamos el hecho de que son objetos y podemos hacer uso del polimorfismo.

```
public enum Operacion {
    SUMA {
        @Override
        public double aplicadaA(double unOperador, double otroOperador) {
            return unOperador + otroOperador;
        }

    },
    RESTA {
        @Override
        public double aplicadaA(double unOperador, double otroOperador) {
            return unOperador - otroOperador;
        }
   },
    ...;

    public abstract double aplicadaA(
        double unOperador, double otroOperador);
}
```



NO A LA SOBREUTILIZACIÓN

Es interesante saber que las enumeraciones permiten realizar la definición de conjuntos de singleton, elementos conocidos bien definidos. Por esto mismo no deben ser utilizados de más, solamente cuando tenga sentido en el dominio del problema. Principalmente porque no son extensibles, ya que son clases finales y objetos globales que generan mucha dependencia.

Dos clases están fuertemente relacionadas al uso de las enumeraciones, la clase **EnumSet** y la clase **EnumMap**. Estas clases están altamente especializadas y optimizadas para trabajar con enumeraciones y vienen a reemplazar antiguas prácticas como utilizar máscaras de bits. La clase **EnumSet** es un **Set** exclusivamente para trabajar con elementos de una enumeración en particular. Posee varios métodos estáticos que facilitan la creación de estos conjuntos. El primero es **allOf(Class)** que permite obtener todos los elementos de una enumeración como un conjunto.

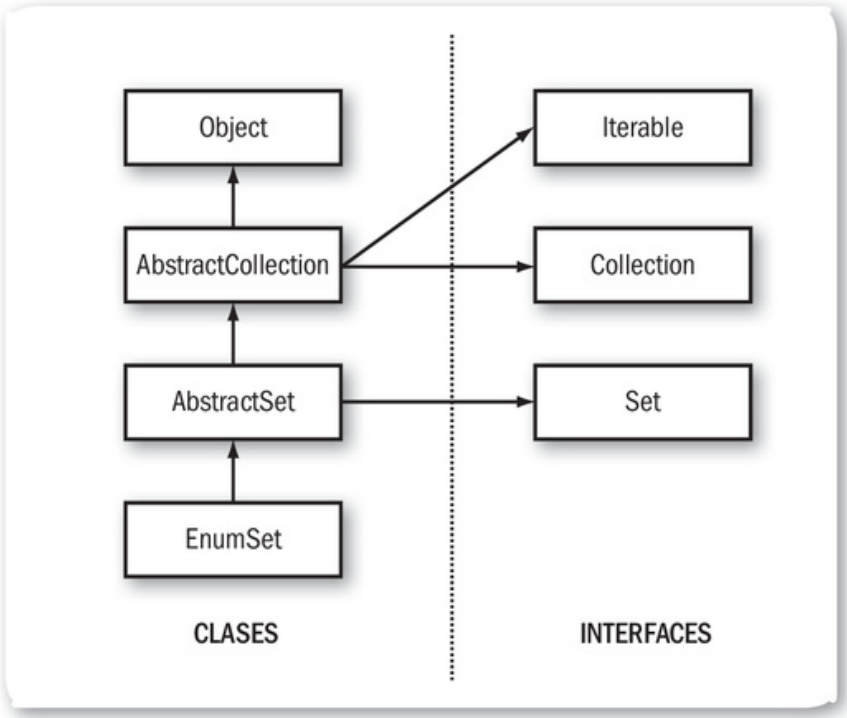


Figura 5. En esta imagen podmeos ver el diagrama que corresponde a la jerarquía de la clase **EnumSet**.



MÁSCARAS DE BITS

En lenguajes antiguos como **C** o **C++**, incluso Java en sus versiones anteriores, hacían uso y abuso de las máscaras de bits. Estas generalmente son un dato de tipo int, donde se utilizan sus bits como indicadores binarios de distintas cosas. Usualmente representaban la combinación de enumeraciones numéricas.

```
Set<Palo> palos = EnumSet.allOf(Palo.class);
// true
assertEquals(4, palos.size());
```

También está el método **noneOf(Class)** que devuelve un set vacío del tipo de enumeración pasado como parámetro.

```
Set<Palo> vacio = EnumSet.noneOf(Palo.class);
// true
assertTrue(vacio.isEmpty());
```

Otro método importante es **of(Enum ...)** que permite crear un conjunto con los elementos especificados.

```
EnumSet<Palo> rojos = EnumSet.of(CORAZON, DIAMANTE);
```

El método complementario a **of** es **complementOf(Enum ...)** y sirve justamente para conseguir el conjunto complemento del especificado.

```
EnumSet<Palo> negros = EnumSet.complementOf(rojos);
// true
assertTrue(negros.equals(EnumSet.of(PICA, TREBOL)));
```

EnumSet provee un método para obtener todos los elementos de una enumeración dentro de un rango. Se basa en el orden de definición.



INFORMACIÓN SOBRE ENUMERACIONES

Si nos dirigimos en nuestro navegador favorito al sitio web **www.javapractices.com** encontraremos un enlace llamado **Type-Safe Enumerations**. Este provee una excelente comparativa entre las enumeraciones antiguas contra las modernas en Java. También veremos cómo lograr enumeraciones seguras en versiones anteriores. Debemos saber que el sitio se encuentra sólo en inglés.

```
// EnumSet con LUNES, MARTES, MIÉRCOLES, JUEVES y VIERNES
EnumSet<DiaDeLaSemana> diasHabiles = EnumSet.range(LUNES, VIERNES);
// true
assertTrue(EnumSet.complementOf(diasHabiles).equals(EnumSet.of(SABADO,
    DOMINGO)));
```

El EnumSet es muy útil cuando tenemos que combinar elementos de enumeraciones. Un ejemplo sería si queremos equipar un auto con distintos componentes.

```
unAuto.equipadoCon(EnumSet.of(AIRE_ACONDICIONADO, STEREO_MP3,
    GPS));
```

Por otra parte la clase EnumMap se utiliza para cuando se quiere relacionar cierta información con los elementos de una enumeración. En **EnumMap** la clave de búsqueda es el elemento distinguido. Un ejemplo sería el menú del plato del día de un restaurante.

```
EnumMap<DiaDeLaSemana, String> platosDelDia =
        new EnumMap<DiaDeLaSemana, String>
        (DiaDeLaSemana.class);
platosDelDia.put(LUNES, "milanesa");
platosDelDia.put(MARTES, "pasta");
platosDelDia.put(MIERCOLES, "pescado");
platosDelDia.put(JUEVES, "sopa");
platosDelDia.put(VIERNES, "asado");
```



CURIOSIDAD SOBRE ENUMMAP

Si bien EnumMap no deja de ser un mapa como cualquier otro, tiene una particularidad que recibe como argumento en la construcción la clase de la enumeración que nos interesa que contenga. Esto puede parecer redundante pero es necesario para que en tiempo de ejecución se verifique que la clave sea un elemento de la enumeración. La información de los genéricos es para tiempo de compilación.

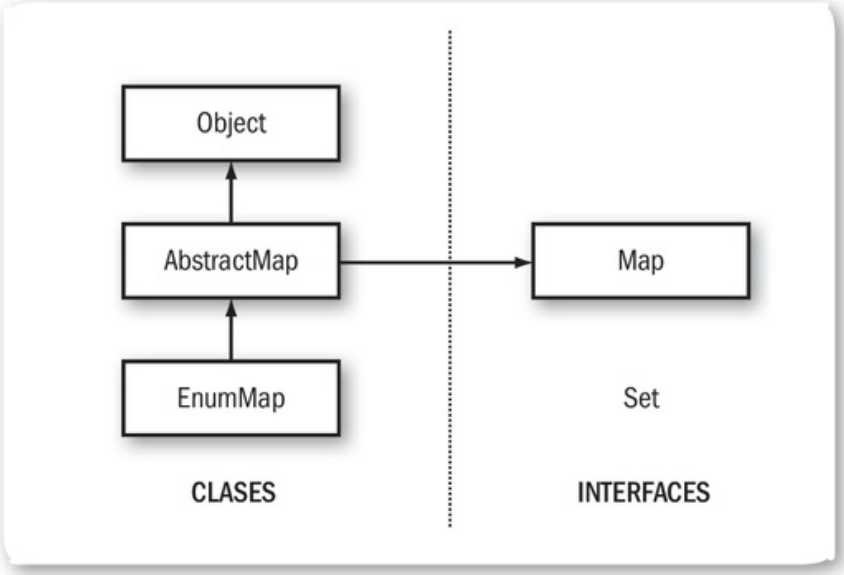


Figura 6. Esta imagen nos muestra una diagrama que representa la jerarquía de la clase **EnumMap**.

RESUMEN

Las enumeraciones fuertemente tipadas son una gran evolución de las viejas e inseguras enumeraciones numéricas o de **String**. Tienen muchas ventajas sobre estas últimas y ninguna de sus desventajas. Son objetos que representan correctamente un concepto del dominio y podemos hacer uso del polimorfismo al darles a cada elemento distinguido un comportamiento distinto. Las clases que acompañan a las enumeraciones, **EnumSet** y **EnumMap**, son de gran utilidad, especialmente **EnumSet** que tiene varios **métodos factory** para facilitar el uso masivo de enumeraciones.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Qué es una enumeración?
2	¿Cuántos elementos puede tener una enumeración?
3	¿Son clases las enumeraciones? ¿Y los elementos?
4	¿Se pueden sobrescribir los métodos name() y ordinal()?
5	¿Pueden las enumeraciones ser extendidas?
6	¿Puede haber enumeraciones abstractas?
7	¿Qué es un EnumSet? ¿Para qué sirve?
8	¿Para qué utilizaría un EnumMap?
9	¿Cuándo utilizaría una enumeración en vez de una constante?
10	¿Se imagina algún mal uso para las enumeraciones?

ACTIVIDADES PRÁCTICAS

1	Trate de hacer que las células sean una enumeración en El juego de la vida.
2	Ahora trate de hacer que los vecindarios sean una enumeración. ¿Tiene sentido tal cambio?
3	Supongamos que queremos agregar una regla que dice que las celdas sólo pueden revivir una vez. ¿Cómo lo implementaría? ¿Sigue teniendo sentido tener las células como una enumeración?
4	Imagine los estados de un vuelo, “a tiempo”, “embarcando” y “demorado”. Ahora suponga que queremos mostrar mensajes en las pantallas del aeropuerto que dependen del estado del vuelo. Implemente.
5	Implemente un mazo de cartas.

*08

Excepciones

Cuando nuestro programa está siendo ejecutado y ocurre alguna situación inesperada, el sistema notifica de este hecho mediante eventos denominados excepciones. Las excepciones son la forma de avisar y detectar errores en la ejecución o cuando ocurre un acontecimiento extraño. En Java las excepciones son objetos y podemos definir nuestra propia jerarquía de excepciones. Aquí las analizaremos.

- Definición 150
- Uso 154
- Excepciones chequeadas..... 157
- Excepciones no chequeadas.. 158
- Errores 159
- Resumen..... 159
- Actividades..... 160

Definición

Es necesario tener en cuenta que una excepción es un evento no esperado que ocurre durante la ejecución de un programa y que interrumpe el flujo normal del este. Cuando ocurre un error durante la ejecución de un método, este crea un objeto con información sobre las causas del error en el contexto de la ejecución. Este es el objeto excepción y es pasado al sistema (a la máquina virtual) para que lo trate. Esto se conoce como lanzar una excepción. Cuando lanzamos una excepción el sistema trata de encontrar algún método en la cadena de llamados que puede manejarla. Para entender cómo funciona esto, en primer lugar tenemos que comprender cómo funcionan las llamadas a métodos, por lo tanto lo analizaremos a continuación.

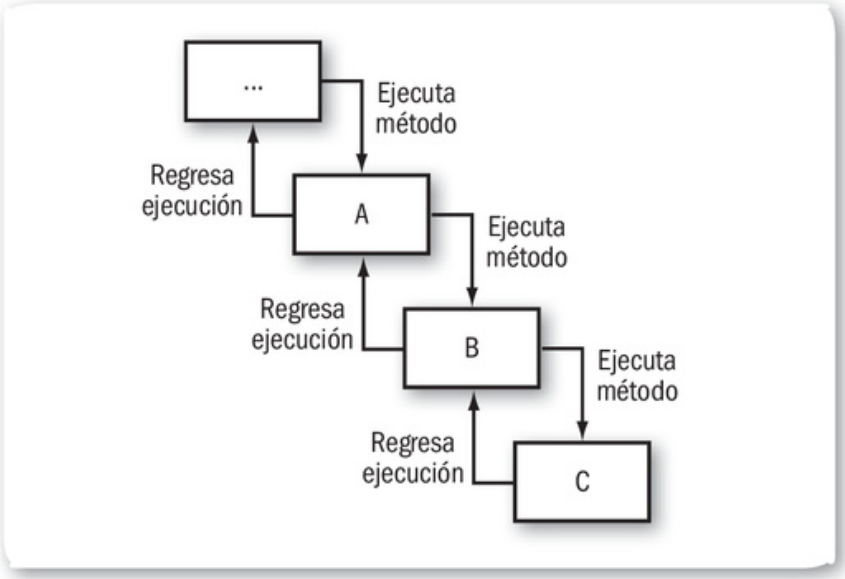


Figura 1. En esta imagen vemos un **`stack`** está formado por invocaciones a métodos, además crece y decrece a medida que se termina la ejecución particular de un método.

Cuando el sistema ejecuta un método asociado a un envío de mensaje (o un método estático), crea una estructura en la memoria con la información relacionada, el método, los argumentos, entre otros datos. Estas estructuras, llamadas activation records o registros de activación, se van enlazando a medida que de un método llama a otro.

De esta forma cuando un método finaliza su ejecución, se continúa con la ejecución del método anterior en la cadena. A esta cadena se la conoce como call stack (pila de llamadas) y viene de la representación en la memoria de los **activation records**. Cuando lanzamos una excepción, el runtime (la máquina virtual **Java**) va buscando en el **call stack**, en orden reverso de ejecución, si en alguno de los métodos hay un **exception handler**, o manejador de excepciones, asociado al tipo de la excepción lanzada.

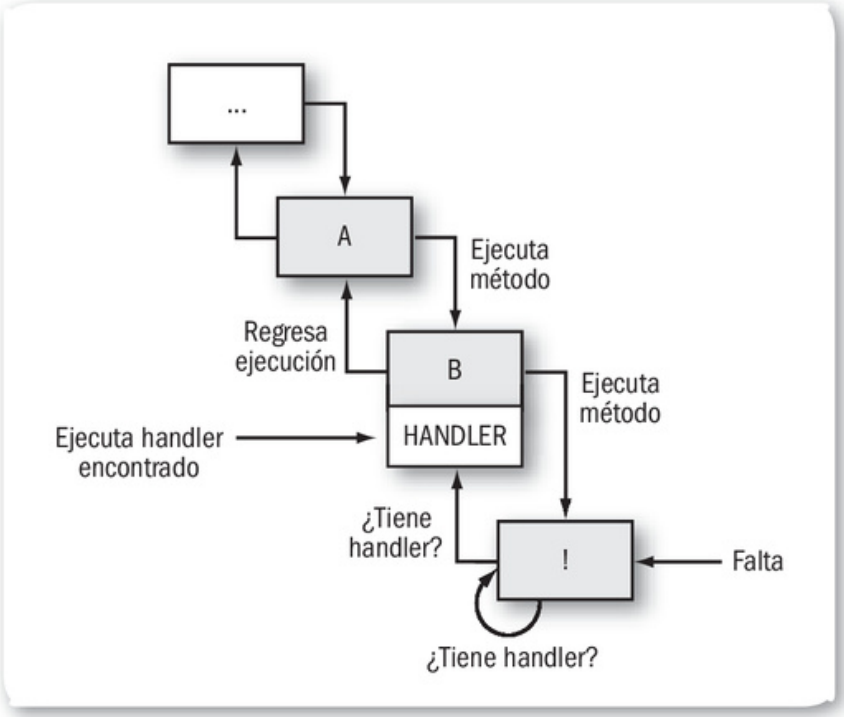


Figura 2. En un `stack` se busca la primera activación con un `handler` adecuado, yendo hacia atrás en la cadena.

CÓDIGOS DE ERROR

Los lenguajes como **C** que no tienen excepciones fuerzan al programador a verificar constantemente los resultados de las operaciones para saber si hubo un error o no. Para esto los métodos regresan un dato que representa un código de error o una marca booleana para indicar éxito o fracaso. El código así escrito sufre de gran cantidad de if y otros enredos para manejar los casos de error.

Recordemos que cuando encuentra el primer handler, se encarga de ejecutar el código asociado y posteriormente regresa la ejecución al método al cual pertenece el **handler** correspondiente, de esta forma se encarga de abortar la ejecución de los métodos del **call stack** que están entre el punto de error y el **handler**.

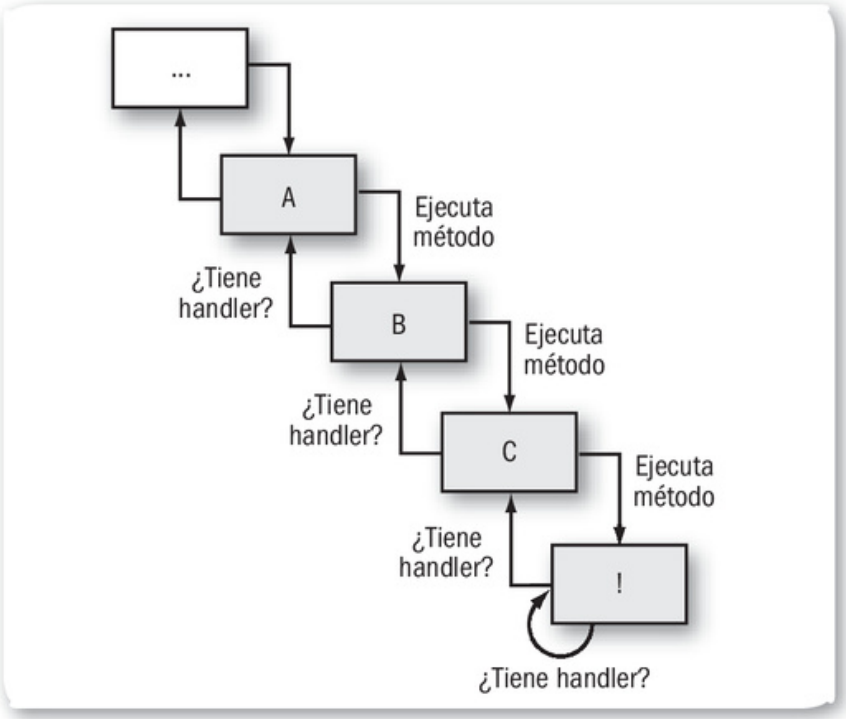


Figura 3. Cuando se busca un **`handler`** en el **`stack`**, si no se lo encuentra se sigue hasta llegar al **`handler`** por defecto, que termina la ejecución del programa.

Cuando el sistema no encuentra ningún **handler** apropiado, ejecuta su propio **handler** existente por defecto, que aborta la ejecución del programa. El hecho de encontrar un manejador apropiado y de ejecutarlo se conoce como **catchear** (de **catch**) o atrapar la excepción.

En Java los objetos que pueden ser lanzados como excepciones son aquellos que son del tipo **Throwable** o de una de sus subclases. Esta clase tiene dos subclases que definen la estructura básica de todas las excepciones y errores en Java. Por un lado tenemos los errores graves de sistema, que no deberían ser nunca de interés a una aplicación, representados por la clase **Error** y sus descendientes. Por otro, las

situaciones excepcionales durante la ejecución de una aplicación, representados por la clase Exception y sus subclases. Es de particular interés la subclase **RuntimeException**, que generalmente representa errores en la lógica del programa que tienen un trato especial en el lenguaje.

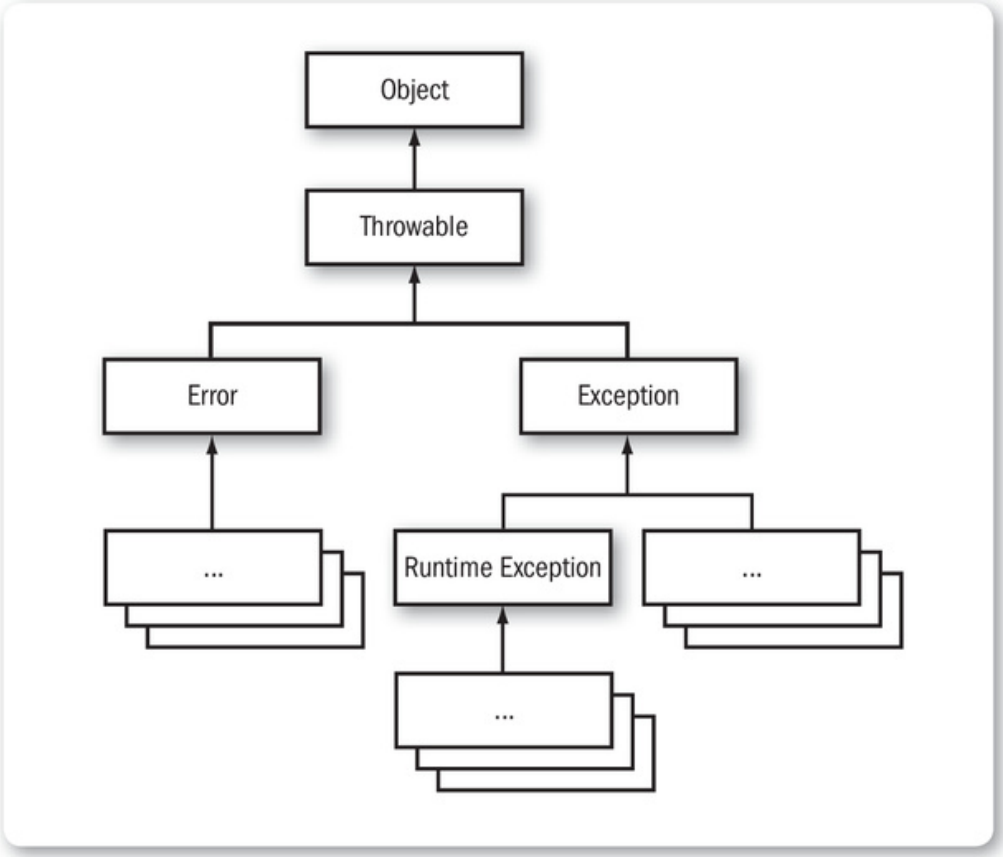


Figura 4. Jerarquía de las excepciones en Java. **`Throwable`** es la clase base para todas las demás. Aquí podemos notar como **`RuntimeException`** hereda de **`Exception`**.

THROWABLE NO ES CHEQUEADA

Las excepciones chequeadas son aquellas que heredan de **Exception**, excepto las que lo hacen de **RuntimeException**. Si deseamos implementar nuestras propias excepciones no chequeadas y no deseamos heredar de **RuntimeException** podemos extender **Throwable**. Pero no solamente podemos crear errores, sino también alguna forma de control de flujo ajena a las normales. No recomendable.



Uso

Las excepciones son bastante sencillas de utilizar. Empecemos por ver cómo lanzamos una excepción.

```
throw new IllegalArgumentException("unParametro");
```

Como vemos, nos encargamos de utilizar la palabra clave throw seguida del objeto excepción que queremos lanzar. Generalmente este es creado en ese mismo momento, como en el ejemplo.

Cuando estamos interesados en manejar excepciones, tenemos que envolver el código que la puede provocar con la declaración **try** y **catch**.

```
try {
... // código que en algún punto puede producir una excepción
} catch(TipoDeExcepcion excepcion) {
... // código para manejar la situación de error
}
```

El código dentro del catch (**handler**) se ejecuta cuando el runtime decide que es el handler apropiado para tratar la excepción. Solamente pueden tratar excepciones que sean del tipo (o un subtipo) del especificado y dentro del bloque en que se tiene acceso al objeto excepción, tal como si fuera un parámetro. Es posible definir varios **handlers** para varios tipos de excepciones en una misma declaración **try**.

```
try {
    ...
} catch(UnTipoDeExcepcion excepcion) {
    ...
} catch(OtroTipoDeExcepcion otraExcepcion) {
    ...
} catch(YOtroTipoMasDeExcepcion yOtraExcepcionMas) {
    ...
}
```

De esta forma se ejecutará solamente el primer **handler** asociado al tipo de excepción que ocurra. Si hay varios candidatos posibles, por ejemplo, si tenemos un **handler** para una subclase de una clase de excepción que también queremos manejar, entonces se ejecuta el primero definido. Por eso los tipos más abstractos se definen por último. Cabe aclarar que dentro del bloque del manejador, solo se puede acceder a las variables declaradas fuera del **try** y al objeto excepción definido en el **catch** correspondiente.

Java provee una gran variedad de excepciones incluidas en el sistema que es recomendable aprender y usar antes de crear nuevas excepciones. Algunas de las excepciones más utilizadas son:

EXCEPCIONES MÁS REUTILIZADAS EN JAVA

EXCEPCIÓN	USO
IllegalArgumentException	Parámetro no nulo inválido.
IllegalStateException	El objeto receptor se encuentra en un estado inválido para ejecutar el método.
NullPointerException	Parámetro nulo.
IndexOutOfBoundException	Parámetro para usar como índice en una colección es inválido.

Tabla 1. Excepciones más reutilizadas en general en los programas Java.

Cuando queremos crear nuestras propias excepciones, lo más recomendable es heredar de alguna de las clases ya existentes dentro de la jerarquía de **Exception**.

Teniendo en cuenta las excepciones ya existentes podremos ver donde es posible ubicar mejor nuestra nueva clase.

NOMENCLATURA

En el mundo Java es una convención que las excepciones tengan como sufijo la palabra **Exception** y que el nombre refleje adecuadamente la situación de error. También es normal facilitar varios constructores que aceptan información sobre la falla, un mensaje de error y finalmente la causa (excepción) del error.

```
public UsuarioNoExistenteException extends Exception {
// el constructor recibe la información de lo ocurrido

    public UsuarioNoExistenteException(String nombre) {...}
// hay métodos para obtener esa información
    public String nombreUsado() {...}
}
```

Es adecuado dotar a nuestra excepción de un constructor que acepte la información para entender la causa del error y, también, proveer métodos para acceder a dicha información en el código de manejo.

Por otra parte, es necesario que tengamos en cuenta que las excepciones pueden ser encadenadas en una relación causal. Para esto podemos utilizar el constructor para pasar la excepción que originó el error como parámetro o utilizar el método initCause.

```
public void ingresarConUsuario(String nombre, String clave) throws In-
gresoNoPermitidoException {
    try {
        ...
    } catch(UsuarioNoEncontradoException e) {
        throw new IngresoNoPermitidoException(e);
    }
}
```

Este anidamiento de excepciones es muy útil para abstraer al cliente de una **API** de errores internos. Abstraemos las excepciones y arrojamos una más significativa a la tarea que representa el método. También logramos disminuir el acoplamiento al reducir el número de excepciones que arroja un método y que forzamos a los métodos clientes a atrapar o arrojar. Por último es aconsejable que cuando lanzamos excepciones, dejemos a los objetos involucrados en un estado utilizable. Con esto nos referimos a que, si manipulamos el estado de un objeto y el método falla, ese estado sea válido en la semántica del objeto, de forma que se le pueda seguir enviando mensajes y este responda adecuadamente.

Excepciones chequeadas

Las excepciones que pertenecen a la jerarquía de **Exception**, salvo las que son **RuntimeException**, son conocidas como excepciones chequeadas. Son chequeadas porque el compilador Java fuerza a que se las trate específicamente. Cuando un método quiere arrojar una excepción de este tipo, debe declararlo en la firma del método. Para esto se utiliza la palabra clave denominada **throws** seguida de las clases de las excepciones que se van a lanzar separadas por coma.

```
public Usuario buscarUsuarioCon(Nombre nombre) throws
    UsuarioNoExistenteException {
        ...
        throw new UsuarioNoExistenteException(nombre);
        ...
}

public void guardar(Object algo) throws ObjectoNoGuardableException,
    ErrorAlGuardarException {
        ...
}
```

El compilador nos alerta si no declaramos las excepciones en la firma del método. Otra opción es atrapar la excepción y tratarla.

Estas excepciones representan situaciones de errores esperables por la aplicación y deberían ser recuperables. Por recuperable entendemos que la aplicación puede tomar acciones para evitar el error en el futuro. Un ejemplo es si nuestra aplicación quiere leer un archivo, le pregunta



CATCHS VACÍOS

Cuando estamos tratando con una excepción chequeada y no queremos arrojarla, debemos tratarla adecuadamente. Adecuadamente significa que debemos hacer algo con ella, no simplemente ignorarla. Ignorar una excepción significa que su código manejador dentro de la estructura catch esté vacío. En general, debemos tratar de evitar fuertemente este tipo de código.

al usuario por el archivo y resulta que no existe, entonces una opción sería preguntar de nuevo. Otro ejemplo es si estamos en un sistema y buscamos un usuario pero este no existe, el sistema podría preguntar si se desea crear el usuario. Estas excepciones representan errores en el dominio del problema y son tratables en la mayoría de los casos.



Excepciones no chequeadas

Las excepciones no chequeadas son aquellas que son de tipo **RuntimeException** o alguna de sus subclases. Estas excepciones son silenciosas, el compilador no nos obliga a atraparlas ni a declararlas en las firmas de los métodos (y por lo tanto no sabemos si un método arroja alguna excepción de este tipo). Estas excepciones representan errores de la aplicación que son recuperables y generalmente resultan de errores de programación o de lógica en el programa. Un caso de estas excepciones son las **NullPointerException**, que aparecen cuando queremos mandarle un mensaje a null.

Resultaría molesto que por todos lados nos obliguen a tratar o declarar estas excepciones que podrían ocurrir en cualquier punto donde se envía un mensaje. Si bien es posible atrapar estos errores y tratarlos, debería ser en casos muy extraños y específicos. En la mayoría de los casos hay que corregir el error en el programa. Las excepciones chequeadas forman parte del contrato de un método, en cambio, las no chequeadas representan qué sucede si se lo rompe.

```
// no es necesario declararla ni atraparla
    public void copiar(String unTexto) {
        if(unTexto == null) throw new IllegalArgumentException("unText");
        ...
}

...
this.copiar(algo);
// no sabemos que arroja una excepción
...
```

Errores

Los errores son problemas que ocurren por fuera de la aplicación y que esta no puede ni anticipar ni tratar, generalmente son problemas que encuentra el runtime con el **sistema operativo**. Por ejemplo, podemos estar leyendo un archivo y se rompe el disco desde el cual estamos leyendo el archivo. Otro ejemplo es cuando el sistema se queda sin memoria para crear nuevos objetos. Recordemos que este tipo de errores no pueden ser manejados por la aplicación y por esta razón terminan con el programa siendo abortado.

Es necesario tener en cuenta que los errores son objetos pertenecientes a la jerarquía de la clase Error y sus subclases.

Debemos tener en cuenta que, en casi todos los casos no tenemos que preocuparnos por este tipo de errores, salvo que estemos desarrollando algún producto crítico o de bajo nivel como un servidor web o alguna herramienta de monitoreo de aplicaciones.

RESUMEN

En este capítulo pudimos conocer de que las excepciones son un poderoso mecanismo para indicar fallas en la ejecución de un programa ya que permiten escribir un bloque de colaboraciones e indicar luego qué se debe hacer ante un determinado error. En Java hay tres grandes grupos de excepciones. Las chequeadas, que son las más comunes y reflejan errores esperables en un dominio. Las no chequeadas, que representan errores en la lógica o uso de un programa. Y, finalmente, los errores, que son problemas más allá del control del programador y de la aplicación.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Qué es una excepción?
2. ¿Cómo se lanza una excepción?
3. ¿Qué ocurre cuando una excepción es lanzada?
4. ¿A qué llamamos **call stack**?
5. ¿De qué forma se atrapa una excepción?
6. ¿Cuáles son las principales clases relacionadas con las excepciones?
7. ¿Qué diferencias hay entre las excepciones chequeadas y no chequeadas?
8. ¿Para qué se utilizan las excepciones no chequeadas?
9. ¿Deben las aplicaciones, en su mayoría, preocuparse de excepciones del tipo **Error**?
10. ¿Es conveniente que una excepción suba varios niveles de abstracción?

ACTIVIDADES PRÁCTICAS

1. Haga una prueba para ver qué información se obtiene en un stack trace. Vea el método **getStackTrace** de **Throwable**.
2. Pruebe cambiar las excepciones usadas en el ejercicio de El Juego de la Vida por algunas propias que sean chequeadas. ¿Cuánto más código tiene que agregar para manejarlas?
3. Busque en la librería Java alguna excepción abstracta. ¿Tiene sentido tal cosa?
4. Codifique un estructura **try/catch** atrapando varias excepciones de modo que algunas pertenezcan a una misma jerarquía. ¿Importa el orden en que se declaran los **catch**?
5. Arroje un **Error**, por ejemplo un **OutOfMemoryError**, y vea qué ocurre.

*09

Genéricos

En versiones de Java anteriores, las clases que tenían que estar preparadas para colaborar con objetos de distinto tipo usaban Object para referirse a ellos. Para solventar esta situación se agregaron los genéricos. Estos transforman las clases en una especie de plantilla, y de este modo evita los molestos cast. En este capítulo aprenderenos su uso.

- Definición 162
- Subtipado 166
- Comodín 166
- Tipos restingidos 167
- Genéricos en el alcance estático 168
- Resumen 169
- Actividades 170

Definición

Lo primero que tenemos que entender para poder comprender cómo funcionan y para qué sirven los genéricos es ver qué problemas existían cuando estos no estaban en el lenguaje.

```
Map menu = new HashMap();
menu.put("ravioles", new Double(14.56));
menu.put("carnes al horno", new Double(32.00));
menu.put("sopa del dia", new Double(7.00));
...
// al no estar tipado puedo poner cualquier cosa
menu.put(new Integer(1), new Auto());
// tampoco sé qué hay adentro
(Float) menu.get("ravioles")
// arroja un error al castear incorrectamente
```

Es necesario recordar que estos son algunos ejemplos de los problemas que generaba el hecho de que no se pudiera especificar el tipo de objetos que contenían las colecciones.

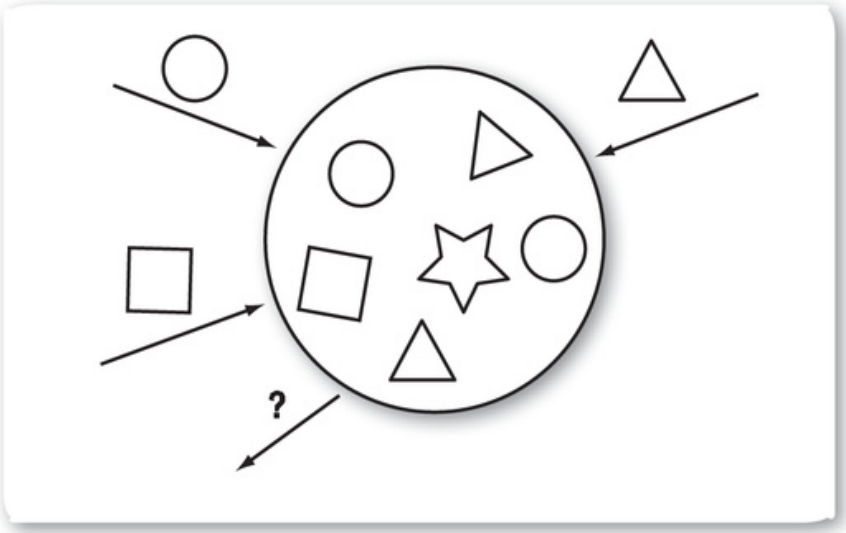


Figura1. Así era el uso de las colecciones antes de la inclusión de genéricos en el lenguaje.

Otro ejemplo que encontramos en la librería básica de Java es la interfaz Comparable. Podemos darnos cuenta de que esta interfaz se encarga de especificar la comparación entre objetos de un mismo tipo de modo de determinar un orden natural entre ellos.

```
// versión vieja
int compareTo(Object o)
// versión nueva
int compareTo(T o)
```

Aquí vemos como el tipo Object fue reemplazado por el tipo **T**, que es un genérico. De esta forma, todo objeto que quiera implementar esta interfaz tiene que especificar a qué tipo corresponde **T**.

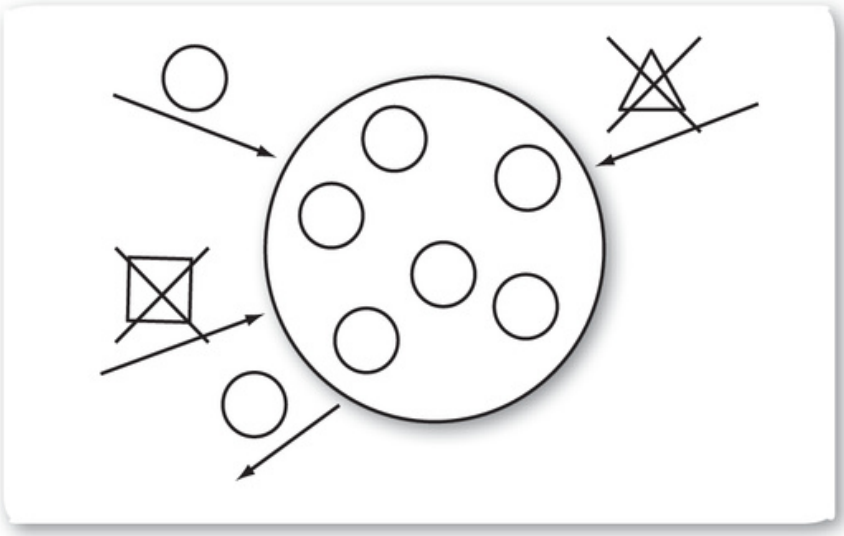


Figura2. Con la incorporación de los genéricos se pudieron tener colecciones genéricas que se especificaban para un tipo determinado, tal como ejemplifica este diagrama.

Se logra que el compilador verifique que los objetos por comparar tengan el tipo adecuado. También evitamos tener que chequear que el objeto pasado como parámetro sea instancia del tipo que nos interesa, así se reducen los códigos y también los errores que se presentan.

Para definir un tipo genérico tenemos que, luego del nombre de clase

o interfaz, enumerar cuántos tipos genéricos vamos a tener y cómo vamos a referenciarlos en el resto del código. Los tipos genéricos se declaran separados por comas entre los símbolos < y >.

```
public interface Comparable<T> {
...
}
```

En el código podemos utilizar el tipo **T** como cualquier otro tipo.

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Generalmente para denominar a los tipos genéricos se usa una sola letra en mayúscula y se utilizan las últimas letras del abecedario comenzando por la letra **T**. También se suele utilizar, cuando hay varios genéricos involucrados, la primera letra del concepto que representan. Por ejemplo **K** para **key** (clave) y **V** para **value** (valor).

Cuando queremos utilizar un tipo genérico (instanciar) tenemos que especificar con qué tipos lo vamos a hacer.

```
    Map<Producto,Precio> catalogo = new LinkedHashMap<Producto,Precio>();
    catalogo.put(jabon, new Precio(2.50));
...
    Precio precio = catalogo.get(jabon);
```



TYPE ERASURE

Los genéricos existen solamente en el código y durante la compilación. No existen en tiempo de ejecución, ya que el compilador borra toda información relacionada con ellos. De esta forma la ejecución es igual que siempre y no hay problemas con aplicaciones antiguas. Igualmente el código resultante es fiable porque el compilador se encargó de verificar que lo sea.

```
...
// error de compilación
    Double precio = catalogo.get(acondicionador);
...
// error de compilación
    catalogo.put("body shower", new Precio(2.35));
```

No solamente podemos especificar tipos genéricos en la definición de un tipo, también podemos utilizar un tipo genérico específicamente para un método. Para ello declaramos el (o los) tipo genérico como modificador del método. Este es válido para el alcance del método, su código, sus parámetros y su tipo de retorno.

```
public class Buscador<T> {
...
    public <U> T[] filtrar(U base) {
        ...
    }
...
}
```

En el ejemplo, el tipo **U** es **inferido** por el compilador y depende del contexto de uso, si pasamos un String entonces **U** será **String**.

Este tipo de uso es muy conveniente para poder forzar el mismo tipo en varios parámetros y al mismo tiempo en el tipo de retorno. Tengamos en cuenta que así hacemos que el compilador nos ayude a forzar una regla del dominio sobre los objetos sin que nosotros trabajemos.



TRABAJANDO SIN GENÉRICOS

Es necesario tener en cuenta que puede ser que estemos trabajando con código viejo y que use los tipos sin especificar el genérico. En estos casos el compilador nos lo hará notar mediante un alerta. Podemos informar al compilador que estamos seguros de lo que hacemos y evitar el alerta mediante la anotación **@SuppressWarnings** con los parámetros unchecked o **rawtypes**.

Subtipado

Una característica que puede resultar extraña es el subtipado de genéricos. Por ejemplo, si tenemos **Habitat<Felino>** y un **Habitat<Tigre>**, no existe relación alguna de subtipado entre estos dos tipos.

```
// inválido
    Habitat<Felino> habitat = new Habitat<Tigre>();
```

Pensemos por qué no tiene sentido que exista un relación. Si el ejemplo anterior fuese correcto, podríamos agregar un puma a un hábitat de tigres. De esta forma, el principio de sustitución, que es la base del subtipado, se ve comprometido.

Comodín

En el uso de los genéricos ocurre que en algunas ocasiones no nos interesa el tipo genérico o específico. Podemos entonces especificar esto con un comodín en lugar del tipo, para ello usamos el símbolo **?**. Este símbolo indica cualquier tipo (aunque no significa que sea igual a **Object**).

```
public Cuidador {
    void llevarAlimentoA(Habitat<?> habitat) {
        ...
    }
}
```



RESTRINGIR MEDIANTE SUPER

Si estamos interesados en aprender sobre el uso y propósito de **super** al restringir tipos en los genéricos, podemos dirigir nuestro navegador a la siguiente dirección: **http://download.oracle.com/javase/tutorial/extra/generics/morefun.html**. Allí encontraremos una breve y clara explicación sobre este complicado modificador. También algunos ejemplos para clarificar este concepto.

Tipos restringidos

Hay veces que queremos que los tipos permitidos para un genérico pertenezcan a alguna jerarquía, con el fin de poder usar ciertos métodos propios a ella en el código genérico. Por ejemplo, si tenemos la clase que representa hábitats en un zoológico y lo hacemos genérico, esperamos que lo que pongamos allí sea un animal y no un número. Por este motivo especificamos que los tipos permitidos tienen que ser **Animal** o uno de sus subtipos. Lo hacemos utilizando la palabra **extends**.

```
public class Habitat<A extends Animal> {
 ...
}
```

En el código de esta clase, todo objeto que sea de tipo **A** será tratado como un **Animal**, pudiendo enviarle mensajes que entiende este tipo.

Por otra parte si queremos que dicho tipo genérico extienda de varios tipos simultáneamente (implementa una o varias interfaces que nos interesan), de esta forma podemos especificarlo al conectar cada tipo que se quiere extender con el símbolo **&**.

```
public class CajonDeVerduleria<V extends Verdura & Fruta> {
 ...
}
```

También podemos utilizar el comodín junto con el extends. Dado que, si tenemos un método para transportar animales de un hábitat



DEFINIR UN PISO PARA LOS TIPOS

Existe también una forma de restringir los tipos utilizando la palabra **super**. Esta permite especificar que aceptamos cualquier tipo que sea un supertipo del especificado (incluido él mismo). Esta restricción es utilizada en casos muy particulares y su concepto es difícil de entender. Recomiendo leer este tema avanzado cuando nos hayamos familiarizado con los genéricos.

a otro, podemos utilizar el comodín con el **extends** para indicar que esperamos un hábitat de algún subtipo de **Animal**, inclusive **Animal**. Si pusiéramos directamente un hábitat de **Animal**, solamente podríamos pasar este tipo de hábitat y no, por ejemplo, un hábitat de tigres.

```
public class Habitat<A extends Animal> {
    void transferirA(Habitat<? extends A> habitat) {
        ...
    }
}
```

Genéricos en el alcance estático

Un uso interesante de los genéricos es cuando son utilizados por métodos estáticos. Veamos un ejemplo concreto.

```
public class Colecciones {
    public static <E> List<E> unaListaCon(E ... elementos) {
        ...
    }

}
```



TIPOS COVARIANTES

En Java, los **array** son **covariantes**. Esto significa que si **B** es un subtipo de **A**, entonces **B[]** es un subtipo de **A[]**. En los genéricos ya vimos que esto no es válido porque podemos cometer errores si asignamos un array de tipo **B[]** a una variable del tipo **A[]** y luego queremos agregar uno tipo **A**. Si bien parece que tendría que cumplirse y que es natural, la covariancia no funciona en los genéricos.

Tenemos un método genérico estático que no tiene ninguna particularidad, salvo cuando lo usamos.

```
import static Colecciones.*;
    ...
        ...
    List<String> nombres = unaListaCon("Pedro", "Juan", "Lucia");
```

Gracias a los genéricos tenemos un método para poder crear listas enumerando sus elementos. El compilador se encarga de realizar todos los chequeos de tipado para que el uso sea correcto. Este tipo de métodos estáticos son muy útiles para la construcción de objetos complejos como en este caso. Escribir uno mismo el código para tal cosa es tedioso, largo, requiere especificar tipos y puede tener errores.

De esta forma tenemos un código más claro, ya que ilustra nuestra idea de que solo se aceptan determinados tipos (y que no esté en un comentario o en el nombre de los argumentos). A su vez, tenemos un código más seguro, dado que no tenemos que realizar las validaciones ya que las hace el compilador.



RESUMEN

Los genéricos fueron un gran aporte al lenguaje Java ya que permitieron mejorar notablemente el código y eliminaron errores debidos a los casteos y al mal uso de las colecciones. Los genéricos permiten escribir código seguro y más claro al hacer evidentes las restricciones de tipo que el programador tiene en su mente. Lamentablemente algunas propiedades de los genéricos en Java son muy complicadas de entender y también poseen algunas limitaciones como el **type erasure**.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Qué son los genéricos?
2. ¿Para qué se utilizan?
3. ¿Cuáles son los problemas que solucionan?
4. ¿Qué significa **type erasure**?
5. Si no quiero especificar ningún tipo, ¿qué utilizo?
6. ¿Cómo puedo restringir los tipos para un genérico?
7. ¿De qué forma es la relación de subtipado con los genéricos?
8. ¿Qué particularidad tiene el uso de los genéricos en el alcance estático?
9. ¿Qué pasa si el compilador no puede resolver correctamente un genérico?
10. ¿Qué otras restricciones se pueden aplicar a los tipos de un genérico en su definición?

ACTIVIDADES PRÁCTICAS

1. Modele una caja en la que se puedan poner cualquier tipo de cosas.
2. Ahora modele una huevera usando la caja.
3. Trate de guardar una manzana en la huevera. ¿Es posible?
4. Modele un zoológico con distintos hábitats para los animales.
5. Agregue la capacidad de mover animales de un hábitat a otro, siempre y cuando alberguen animales de la misma familia.

*10 –

Librería base

Conocer las herramientas disponibles para realizar un trabajo es fundamental, ya que permite determinar cuál se adecua mejor a la tarea. Por este motivo, toda persona que desea aprender Java debe conocer las clases e interfaces que se proveen desde la instalación. En este capítulo analizaremos los recursos con los cuales contamos.



- **Librería y objetos básicos 172**
  - java.lang.Object 172
  - java.lang.Boolean 174
- **Colecciones 177**
  - java.util.Collection 178
  - java.util.Set 180
  - java.util.Map 181
- **Ejercicio: colecciones diferentes 182**
- **Clases útiles 186**
  - Ejercicio: alternativa a java.util.Date 188
- **I/O 192**
  - java.io.InputStream y su familia 192
  - java.io.Reader y java.io.Writer 195
- **Resumen 197**
- **Actividades 198**

Librería y objetos básicos

Java posee una amplia librería base que viene con la instalación inicial, ya sea del kit de desarrollo o del entorno de ejecución. Esta librería provee lo necesario para desarrollar todo tipo de aplicaciones, desde las de escritorio hasta aplicaciones web. Permite crear ventanas, escribir archivos, conectarse a un servidor y muchas cosas más.

De esta forma comenzaremos nuestra exploración de la librería por las clases que conforman los ladrillos básicos sobre los cuales se construyen las demás clases del sistema.

java.lang.Object

Esta clase, como ya sabemos, es la clase padre que corresponde a Java, de la cual todas las demás heredan. Aquí encontramos definidos los métodos más primitivos y generales que podemos esperar de los objetos. Lo más básico que podemos saber de un objeto en Java es lo siguiente: si es igual a otro objeto (**equals**), a qué clase pertenece (**getClass**) y acceder a una representación en texto (**toString**).

```
Class<?> getClass()
boolean equals(Object otroObjeto)
String toString()
```

En este sentido, otro método muy importante, pero que a primera vista no nos transmite su utilidad es el llamado **hashCode**. El **hash code** es un numero entero (**int**) que es usado por ciertas colecciones, generalmente los mapas (diccionarios), para guardar el objeto de forma que el **hash code** sea el índice de este.

```
int hashCode()
```

Finalmente **Object** provee otros métodos que no son muy utilizados, al menos en la gran mayoría de las situaciones. Uno de ellos es el método para clonar (duplicar) un objeto, el cual no es usado, ya que la clase de los objetos por clonar tiene que implementar la interfaz

Cloneable y tener ciertas consideraciones. Los demás métodos definidos son para sincronizar, de forma rudimentaria, varios hilos de ejecución. Estos métodos no deberían ser utilizados normalmente y se recomienda usar otras maneras de sincronización más avanzadas.

Implementar los métodos equals y hashCode

Los métodos **equals** y **hashCode** son especiales y requieren cuidado a la hora de ser sobrescritos en las otras clases.

El método **equals** requiere que se implemente una relación de **equivalencia** que debe ser re□exiva, simétrica y transitiva. Es re□exiva si para todo objeto no nulo **x**, **x.equals(x)** retorna **true**. Es simétrica si se cumple que para todo par de objetos no nulos **x** e **y**, **x.equals(y)** es igual **y.equals(x)**. Finalmente, es transitiva si, y solo si, para los objetos no nulos **x**, **y** y **z**, se cumple que si **x.equals(y)** regresa **true** a su vez que **y.equals(x)** también lo hace, entonces **x.equals(z)** tiene que regresar **true**. También se espera que el método sea consistente, esto quiere decir que sucesivas invocaciones del método con los mismos parámetros arrojen los mismos resultados (a menos que el objeto cambie). Finalmente, todo objeto **x** no nulo es distinto a **null** y por lo tanto **x.equals(null)** da **false**.

> EQUALS Y HASHCODE REQUIEREN CUIDADO A LA HORA DE SER SOBREESCRITOS EN OTRAS CLASES

El método **hashCode** tiene como objetivo utilizar la información del objeto (sus atributos) para generar un número entero utilizado en las colecciones que usan **tablas de hash** internamente. La idea es que utilizando los atributos que definen la identidad del objeto



STREAM DESDE STRING

Los **stream** ofrecen una abstracción a la transmisión de bytes y no están fijos a la red o a los archivos. Podemos crear **stream** para cualquier cosa. Una de las cosas más comunes es querer obtener un **InputStream** a partir de un **String**. Lamentablemente, debemos tener en cuenta que Java ofrece una clase, la **StringBufferInputStream**, que es actualmente depreciada. La opción es entonces utilizar un **ByteArrayInputString** usando los bytes del texto o directamente un **StringReader**.

(generalmente son los mismos que se utilizan en el **equals**) se calcule un número que sea bien distinto al generado por otro objeto ligeramente parecido. Objetos iguales deben devolver el mismo número, pero los objetos que devuelvan el mismo número no tienen que ser necesariamente iguales (pensemos que siempre habrá objetos distintos con el mismo número cuando hay más de 4294967295 objetos). Este cálculo debe ser rápido y consistente mientras el objeto no cambie. Recomendamos leer más sobre el tema de **tablas de hash**, ya que es complicado pero muy interesante.

java.lang.Boolean

La clase **Boolean** modela los valores de verdad y falsedad booleanos y en sus instancias encapsula los valores primitivos **boolean**, **true** y **false**. Provee métodos estáticos para pasar de **String** a **boolean** y viceversa.

```
static String toString(boolean b)
static Boolean valueOf(boolean b)
static Boolean valueOf(String s)
static boolean parseBoolean(String s)
```

java.lang.Number, hijos y Java.lang.Math

Number es la superclase de todos los tipos de números que hay en Java. Esta superclase solamente define métodos que permiten obtener los distintos valores primitivos a partir de un objeto numérico.

```
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()
```

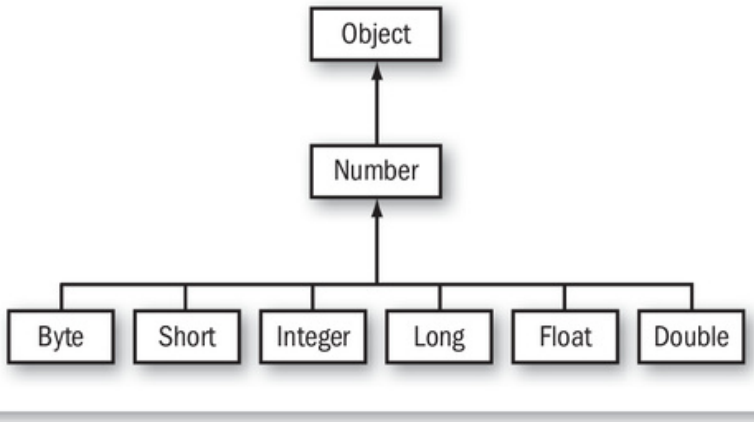


Figura 1. En este diagrama podemos apreciar la jerarquía que corresponde a **Number** y los distintos tipos de números.

Sus subclases más conocidas son aquellas asociadas a los tipos numéricos primitivos, **Integer**, **Byte**, **Short**, **Float**, **Long** y **Double**. Estas clases proveen métodos para manipular los números.

Estas clases no son para enfrentar la realización de operaciones numéricas sino que simplemente sirven para envolver los tipos primitivos y de esta forma permitirles interactuar como si fueran objetos (podemos ver un ejemplo claro de esto en las colecciones).

Por su parte, la clase **Math** es un compendio de los métodos abstractos que implementan la mayoría de las funciones matemáticas que encontramos en cualquier calculadora científica. Primero, posee las constantes matemáticas **Pi** y **E** y, luego, ofrece métodos para las operaciones trigonométricas, para hacer cálculos exponenciales, redondeos y obtener números aleatorios, entre otros más.

```
static double E = 2.718281828459045d;
static double PI = 3.141592653589793d;
static int min(int a, int b)
static double sin(double radianes)
static double toRadians(double grados)
static double random()
static double log(double a)
static double exp(double a)
```

java.lang.String y Java.lang.Character

Character es la clase asociada con los **char** y define los caracteres como objetos. Java soporta los caracteres **Unicode** y soporta todas sus representaciones. La clase provee varios métodos para caracteres a partir de algún código **Unicode**, varios métodos para categorizar al carácter y para pasar de minúscula a mayúscula y viceversa.

Los **String** representan cadenas de caracteres y tienen su forma literal tipo **"abc"**. Son inmutables, o sea que sus valores (los caracteres) no pueden cambiar con el tiempo, y cualquier método que opere sobre ellos devuelve un nuevo **String**. Esta clase tiene la mayoría de los métodos que necesitamos para manejar cadenas de caracteres. Algunos que mencionaremos a continuación son para saber su longitud, conseguir una porción de él, buscar pedazos de texto dentro de la cadena, concatenar más texto y reemplazar secciones. Conviene revisar la documentación para conocer todas las funcionalidades provistas.

```
char charAt(int indice)
String concat(String texto)
boolean equalsIgnoreCase(String texto)
boolean isEmpty()
int length()
replace(String expresionRegular, String reemplazo)
```

Hay dos clases asociadas a los **String**, **StringBuilder** y **StringBuffer**, que se utilizan cuando se quiere construir o modificar **String**s de gran tamaño de forma eficiente en cuanto a memoria y a velocidad. Las operaciones principales de estas clases son insertar (**insert**) y agregar (**append**). Veremos estas clases en detalle más adelante, en este mismo capítulo.



EXPRESIONES REGULARES

Las expresiones regulares, también conocidas como **regex**s, son una notación para definir pequeños lenguajes o patrones. Generalmente se escriben como un **String** y son muy versátiles para realizar búsquedas y verificaciones de forma en textos. Java ofrece un muy buen soporte para **regex**. Si vemos la documentación asociada a la clase **java.util.Pattern**, tendremos una breve introducción a este tema.

Colecciones

Debemos entender que las colecciones son elementos fundamentales de cualquier programa, ya que permiten manejar agrupaciones de objetos de forma simple y consistente. Existen distintos tipos de colecciones que modelan de forma distinta el agrupamiento de los objetos y tienen propósitos bien distintos. Es conveniente conocerlas bien para poder elegir la mejor para cada tarea.

Comenzaremos por las clases e interfaces más simples.

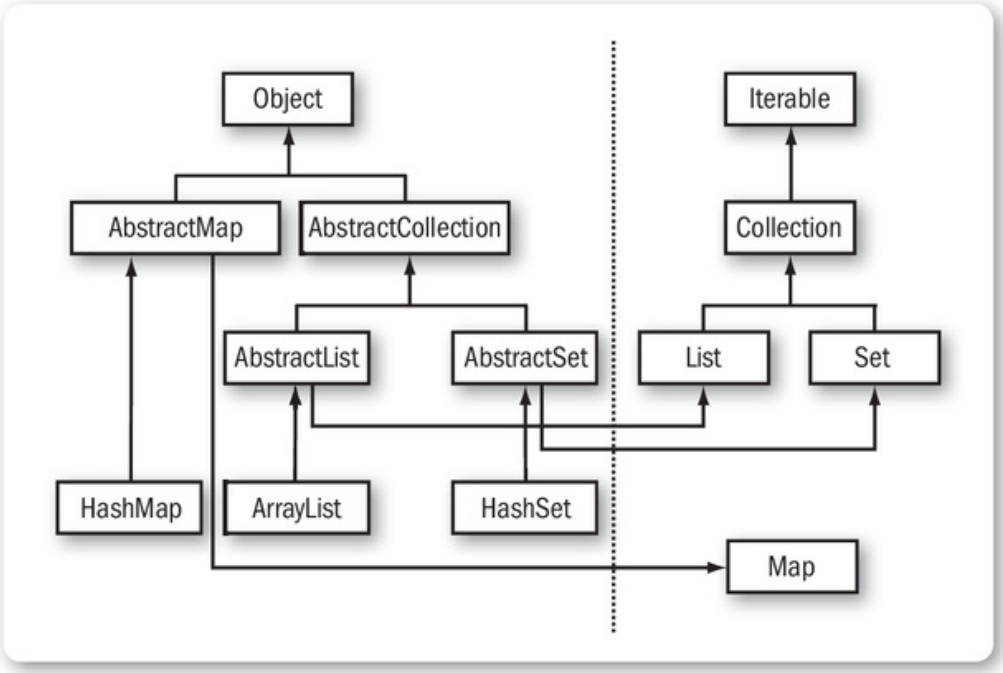


Figura 2. En este diagrama podemos apreciar la jerarquía de las distintas colecciones que podemos encontrar en Java.

GOOGLE GUAVA

Si nos dirigimos a **http://code.google.com/p/guava-libraries** visitaremos el sitio de este proyecto de **Google** de características similares al **Commons** de **Apache**. El proyecto incluye una enorme cantidad de implementaciones de colecciones eficientes para distintos tipos de uso. Así mismo contiene métodos auxiliares para trabajar con los tipos primitivos, con archivos y con la red.

java.util.Iterable y Java.util.Iterator

Estas interfaces permiten que las agrupaciones de objetos (por ejemplo las colecciones) sean recorridas de a un elemento por vez. Todas las colecciones implementan **Iterable**, que permite que sean utilizables en la estructura **for each**. Todo **Iterable** da acceso a un **Iterator**, que es un objeto encargado de realizar el recorrido de los objetos. También permite remover un elemento mientras se realiza el recorrido sin incurrir en un error. Debemos saber que no permite hacer inserciones o agregar objetos nuevos mientras se recorre.

Veamos el protocolo ofrecido por **Iterable**.

```
Iterator<T> iterator()
```

Y a continuación el de **Iterator**.

```
boolean hasNext()
T next()
void remove()
```

Siempre hay que llamar a **hasNext()** antes que a **next()** para asegurarse de que hay más elementos para ver y no obtener un excepción.

java.util.Collection

Esta es la interfaz raíz de todas las colecciones en Java (excepto de los diccionarios o mapas) y modela las agrupaciones de objetos sin especificar si estos tienen un orden o no, o si se acepta el mismo objeto



MÉTODOS AUXILIARES PARA ARRAY

Al igual que tenemos los métodos de la clase **Collections**, Java nos ofrece la clase **java.util.Arrays**. Dicha clase se caracteriza por brindar métodos estáticos para ordenar, buscar, comparar, copiar y crear **array**s de una manera fácil. Estos métodos tienen versiones sobrecargadas para los **array**s de tipos primitivos así como también versiones genéricas para los objetos.

varias veces o, incluso, soporta **null** entre sus elementos. Lo que sí define es un protocolo mutable para colecciones, pero deja claro que los métodos para realizar los cambios son opcionales de implementar (arrojando **UnsupportedOperationException**). Claramente este tipo de interfaz no es muy buen diseño, ya que puede llevar a obtener un error en la ejecución. Hubiera sido mejor separar esta interfaz en dos o más, una interfaz para los protocolos inmutables y otra para los mutables. De esta forma se podría definir mejor qué tipo de colección se ofrece y qué tipo se requiere. **Collection** extiende **Iterable**.

> LOS MÉTODOS PARA REALIZAR CAMBIOS SON OPCIONALES DE IMPEMENTAR

Los métodos ofrecidos permiten agregar elementos, removerlos, saber si están o no en la colección, conocer la cantidad de elementos presentes y conseguir un iterador para recorrerlos.

```
    boolean add(T elemento)
//devuelve true si se agregó
    void clear()
    boolean contains(Object o)
    boolean isEmpty()
    Iterator<T> iterator()
    boolean remove(T elemento)
//devuelve true si se remueve
    int size()
    Object[] toArray()
```

java.util.List

Las listas son colecciones que disponen sus elementos por orden de llegada. Es una colección ordenada, también conocida como secuencia. Los elementos tienen un índice numérico, como los **array**, y es posible agregar elementos en cualquier posición. El acceso a ellos se realiza mediante dicho índice, donde el primer elemento tiene índice 0. Las listas generalmente soportan que estén duplicados y permiten realizar búsquedas sobre sus elementos. **List** ofrece un **iterador** distinto

(además del común a todas las colecciones) que permite no solo remover, sino también agregar y reemplazar elementos, además de ofrecer navegabilidad hacia atrás.

```
    void add(int índice, T elemento)
    T get(int índice)
    int indexOf(Object o)
    int lastIndexOf(Object o)
    ListIterator<T> listIterator()
    T remove(int índice)
//devuelve el elemento removido
    T set(int índice, T nuevo)
//devuelve el viejo elemento
    List<T> subList(int desde, int hasta)
```

java.util.Set

El **Set** representa un conjunto de elementos. Esto quiere decir que es una colección que no acepta duplicados y que no define un orden para los elementos que contiene. Esta interfaz no define ningún protocolo extra al de **Collection**, pero especifica la semántica de ellos (usando documentación, lamentablemente). Las implementaciones de **Set** se aseguran que solamente se agregue una vez un objeto determinado, generalmente usando **equals** y **hashCode**, pero solo en el momento de agregarlo. Si un objeto que pertenece a un **set** muta (cambia sus atributos), no se asegura que se mantenga el invariante de que dicho **set** no contiene dos elementos iguales. Por lo tanto hay que asegurarse de utilizar objetos inmutables o de no cambiarlos mientras se los utiliza en este tipo de colecciones.



JAVA FUNCIONAL

En **http://functionaljava.org** encontraremos un interesante proyecto que tiene como objetivo traer conceptos del paradigma funcional a Java. Algunos de los objetos involucrados modelan funciones y colecciones con métodos para mapear y filtrar, entre otras operaciones. Las implementaciones que ofrece de estas son inmutables y configurables (por ejemplo, cómo consideran la igualdad).

java.util.Map

Esta interfaz representa un conjunto de asociaciones entre pares de objetos, donde uno de ellos es considerado como clave. Por lo común se denomina a este tipo de colecciones como diccionarios o mapas y, extrañamente no hereda de **Collection**. Los objetos usados como claves conforman un conjunto y por lo tanto no pueden usarse para asociar dos objetos distintos al mismo tiempo en el mismo mapa (y, al igual que con los **Set**, deberían ser inmutables). **Map** tampoco extiende **Iterable**, por lo tanto los objetos de este tipo no pueden ser usados en el **for each**. **Map** ofrece tres modos de acceder a su contenido, como si fuera una colección. Primero, podemos acceder a las asociaciones mediante el mensaje **entrySet**, que devuelve un **Set<Map.Entry<K,V>>**, o sea un conjunto de pares clave-valor. Segundo, para acceder al conjunto de claves podemos enviarle al mapa el mensaje **keySet**. Finalmente, si solamente nos interesan los valores y no las claves, podemos utilizar el método **values**, que retorna una colección con los valores.

> MAP OFRECE TRES MODOS DE ACCEDER A SU CONTENIDO, COMO SI FUERA UNA COLECCIÓN

Para agregar elementos a un mapa se utilizan los métodos **put** y **putAll**, donde el primero espera como parámetros la clave y el valor, mientras que el segundo espera otro mapa. Para remover elementos se utiliza el método **remove**, que recibe la clave. Este método remueve la asociación completa, clave y valor. Si queremos obtener el valor asociado a una determinada clave, usamos **get** que toma como argumento la clave y devuelve el valor asociado. Si no está asociada la clave a ningún valor en el mapa, se retorna **null**. Podemos consultar si existe una clave en particular mediante **containsKey** y un valor mediante **containsValue**. Extrañamente el método **remove** acepta cualquier objeto



MÉTODOS AUXILIARES DE COLECCIONES

Es necesario comprender que la clase **java.util.Collections** es un repositorio de métodos estáticos que permiten realizar operaciones sobre las colecciones que no están disponibles. Por ejemplo, ofrece métodos para buscar los elementos máximos y mínimos, y ordenarlos. También permite ampliar las colecciones y hacerlas inmutables o sincronizables.

como clave para remover la asociación, pero si no es del tipo de clave apropiado arroja una excepción. Sería más correcto evitar este error forzando la firma del método a que la clave fuera del tipo correcto.

```
V put(K clave, V valor)
//regresa el valor asociado anterior o null si no había
void putAll(Map<K,V> mapa)
void clear()
boolean isEmpty()
int size()
Collection<V> values()
Set<Map.Entry<K,V>> entrySet()
Set<K> keySet()
boolean containsKey(K clave)
boolean containsValue(V value)
V get(K clave)
V remove(Object o)
//devuelve el valor removido o null si no existe tal asociación
```

La interfaz **Entry** está definida en el contexto de **Map,** ya que está íntimamente relacionada a ella. Este es un uso claro para definir interfaces (o clases) anidadas (y estáticas, ya que es pública).



Ejercicio: colecciones diferentes

En este ejercicio queremos obtener unas colecciones que permitan, fácilmente recorrerlas, filtrarlas y manipular sus elementos. Las colecciones actuales de Java sufren del problema que solo delegan todo el trabajo al cliente. Esto hace que se escriba mucho código repetitivo y propenso a errores. Lo que buscamos es que las propias colecciones se encarguen del código repetitivo. También buscamos que sean configurables, por ejemplo que el ordenamiento y la semántica de la igualdad correspondiente sean parametrizables.

Podemos empezar con las listas, las colecciones más sencillas. ¿Qué necesitaríamos de una lista? Inicialmente podemos pedir que se puedan transformar los elementos de esta y obtener otra lista con estos elementos transformados. Esta operación se conoce como **map**.

```
@Test public void testMap() {
    final List<Integer> numbers = list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    final List<String> numbersAsStrings =
        numbers.map(new Fn<Integer, String>() {
            @Override public String f(final Integer i) {
                return i.toString();
            }
    });

    assertEquals(10, numbersAsStrings.size());
    assertEquals("2", numbersAsStrings.at(2));
}
```

Siguiendo con el espíritu de la interfaz **Iterable**, crearemos una interfaz **Mappable** para representar la idea de que un objeto responde a **map**.

```
public interface Mappable<T> {

    <R> Mappable<R> map(final Fn<T, R> fn);

}
```



APACHE COMMONS

Apache tiene bajo este proyecto que se encuentra en **http://commons.apache.org** una gran cantidad de librerías. Estas librerías dan muchas funcionalidades, ya sea para colecciones como para tratar con los tipos primitivos. También ofrece validadores, parsers, clases auxiliares para I/O y mucho más. Definitivamente un lugar que debemos tener en cuenta para cualquier desarrollo.

Para completar, el tipo representa el código que realiza la transformación, simplemente una función (como se entiende en la matemática). En este caso representa las funciones de una variable.

```
public interface Fn<T1, T2> {

    T2 f(final T1 t);
    <T3> Fn<T1, T3> o(final Fn<? super T2, T3> g);

}
```

Podemos extender este concepto aún más, con más variables y más funcionalidades, por ejemplo componer dos o más funciones en una.

```
@Test public void testFnComposition() {
    final Fn<Object, String> stringfy = new
        AFn<Object, String> () {
        @Override public String f(final Object t) {
            return t.toString();
        }};
    final Fn<Integer, Integer> multiplyBy2 = new
        AFn<Integer, Integer> () { @Override public
        Integer f(final Integer i) {
            return i * 2;
        }};
    assertEquals("4", multiplyBy2.o(stringfy).f(2));
}
```



MODELANDO CON COLECCIONES

Al diseñar tenemos que lograr un isomorfismo entre el dominio y lo que programamos. Por este motivo las colecciones no se tienen que utilizar para modelar conceptos, sino como implementación soporte de tal modelo. Por ejemplo, el menú de un restaurant no es simplemente una asociación de un plato con un precio. Diseñar así lleva a poner lógica de manipulación equivocadamente en otros objetos.

Debemos tener en cuenta que esta habilidad de composición de funcionalidad sumada a las operaciones sobre las colecciones brinda una gran flexibilidad a la hora de codificar.

Para continuar, sigamos agregando funcionalidades. Ahora vamos por la conocida operación denominada **foldLeft**. Esta operación permite iterar sobre los elementos e ir acumulando resultados en otro objeto que se va pasando en la iteración.

```
@Test public void testFoldLeft() {
    final List<Integer> numbers = list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    final Fn3<Integer, Integer, Integer> add = new
        AFn3<Integer, Integer, Integer>() { @Override public Integer f(final
            Integer i, final Integer u) {
                return i + u;
    }};

    assertEquals(55, numbers.foldLeft(0, add).intValue());
}
```

La interfaz para esta operación es:

```
public interface Foldable<T> {
    <S> S foldLeft(final S initialValue, final Fn3<T, S, S> fn);
}
```

Debemos tener en cuenta que otras operaciones útiles que podemos definir son **filter** (que podemos utilizar para filtrar elementos de una colección), **zip** (útil cuando necesitamos realizar la ejecución de agrupar de a pares los elementos de dos colecciones) y **for each** (para ejecutar una acción por cada elemento).

```
<S> Zippable<Pair<T, S>> zip(final List<S> list);
void forEach(final A<T> action);
Filterable<T> filter(final P<T> p);
```

Clases útiles

Debemos tener en cuenta que existen muchas clases que pueden sernos útiles. En esta sección conoceremos algunos ejemplos.

java.util.Date y Java.util.Calendar

Java combina la noción de fecha y hora en una sola clase llamada **Date**. Esta clase existe desde la primera versión de Java, en la actualidad es necesario utilizar un calendario, clase **Calendar**, para poder crear y manipular instancias de **Date**. Entre algunos de los problemas que tiene **Date** están la mutabilidad y la no internacionalización. Es ecesario tener en cuenta que igualmente esta clase es ampliamente utilizada en librerías y productos, ya que forma parte de la librería base. A continuación podemos ver un claro ejemplo del uso de **Date** con **Calendar**.

```
// para conseguir la fecha actual
Date ahora = new Date();
// o sino
Date ahora = Calendar.getInstance().getTime();
```

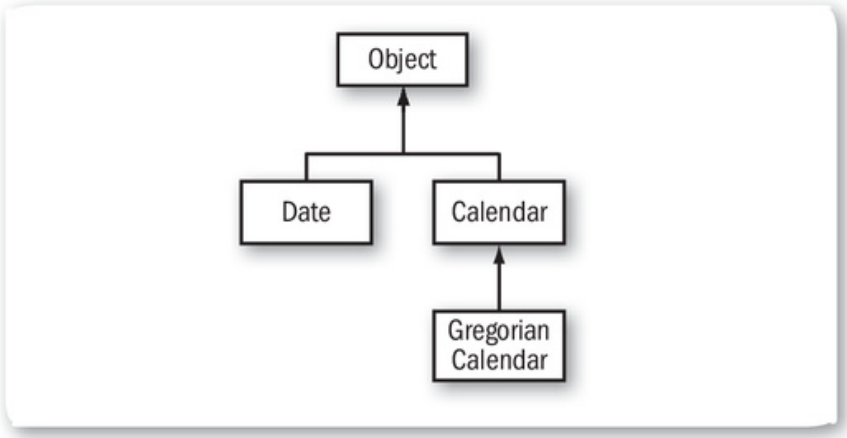


Figura 3. En esta imagen podemos ver un digrama que representa la jerarquía de **Date** y **Calendar**.

Tengamos en cuenta que la recomendación es no utilizar nunca estas clases ya que no están bien diseñadas y su uso es difícil y engorroso. Conviene utilizar alguna librería específica para manejar fechas y horas. Existen algunas como **Joda Time** y **Date4J**.

java.lang.StringBuilder y java.lang.StringBuffer

En Java es muy sencillo concatenar **String**s gracias al operador **+**. Lamentablemente hacer uso intensivo de esta concatenación para generar texto es ineficiente. Recordemos que los **String**s son inmutables y cuando queremos concatenarlos estamos creando todo el tiempo nuevos objetos que consumen mucha memoria.

```
String a = "Hola" + nombre + ". Buen dia!" + pregunta + ". Hoy es " + new
    Date() + ".";
```

¿Cuántas cadenas de carácter se crean en el código anterior? ¿Cuatro, siete o trece? La respuesta es trece (al menos), ya que cada vez que usamos la concatenación estamos creando un **String** (aunque en este caso no nos interesen los intermedios). Para poder realizar concatenaciones que no impacten en la velocidad y en la memoria se crearon dos clases especiales, ellas son **StringBuilder** y **StringBuffer**. Ambas permiten realizar las mismas operaciones de texto con la diferencia de que la primera es una instancia que no puede ser utilizada desde distintos hilos de ejecución, mientras que la segunda (la más vieja de las dos implementaciones) es segura de usar pero un poco más lenta.



JODA TIME

Una librería muy conocida en el mundo Java como alternativa al Date es Joda Time, que podemos encontrar en la dirección **http://joda-time.sourceforge.net**. Esta librería ofrece, además de manipular fácilmente fechas y horas, abstracciones de intervalos de tiempo, duraciones y distintos calendarios. Forma parte de un conjunto de librerías más grande y útil. Es recomendado su estudio.

Las operaciones más importantes son la concatenación (**append**) y la inserción (**insert**). Debemos saber que también proveen métodos para reemplazar y borrar porciones de texto.

```
StringBuilder builder = new Stringbuilder();
builder
        .append("Hola")
        .append(nombre)
        .append(". Buen dia!")
        .append(pregunta)
        .append(". Hoy es ")
        .append(new Date())
        .append(".");
String a = a.toString();
```

Ahora se crean menos **String**s, ocho contra trece.

Ejercicio: alternativa a java.util.Date

Vimos los problemas que tiene la clase **Date** y lo engorroso que es utilizar **Calendar** para manipular las fechas. El propósito de este ejercicio es modelar las fechas de forma distinta. Como premisa, tenemos que obtener un diseño donde las fechas sean inmutables y válidas desde su creación. Segundo, las entidades (conceptos) deben estar modeladas, por ejemplo, al pedirle a una fecha el mes, nos tiene que devolver un objeto que represente al mes en concreto y no un número que sea un índice arbitrario. Tercero, para simplificar no tendremos en cuenta las distintas zonas horarias. Cuarto, separaremos el concepto de fecha del de hora. Por último, obviamente usaremos **TDD** para desarrollar. Manos a la obra.

> SABEMOS QUE ES ENGORROSO USAR CALENDAR PARA MANIPULAR LAS FECHAS

Empecemos por modelar los años. ¿Qué podemos decir de los años? En principio utilizamos un número para nombrarlos y no existe año cero. Después hay años que son bisiestos, donde febrero tiene un días más.

A continuación volcamos estas ideas en algunos test:

```
@Test(expected=IllegalArgumentException.class)
public void testYearZero() {
    Year.number(0);
}

@Test public void testLeapYear() {
    final Year y2k = Year.number(2000);

    assertEquals(y2k.number(), 2000);
    assertTrue(y2k.isLeap());
    assertEquals(y2k.numberOfDays(), 366);
    assertEquals(y2k.february().numberOfDays(), 29);
}
// el test para año no bisiesto es parecido por lo
// cual lo obviamos aquí
```

Ahora bien, hasta este punto ya tenemos una buena aproximación de lo que podemos esperar inicialmente de un objeto que modele un año. Para continuar, sigamos con los meses. Tengamos en cuenta que existen doce meses, algunos tienen treinta días, otros treinta y uno, y está febrero que dependiendo del año tiene veintiocho o veintinueve. Con el mes solo, no podemos saber cuántos días tiene, necesitamos un año en particular. Además, normalmente a los meses también se los identifica con su orden en el año. Para hacer frente a esta situación, escribamos los test correspondientes.

```
@Test public void testMonth30() {
final Month september = Month.september();
final Year y2k = Year.number(2000);

    assertEquals(september.name(), "September");
    assertEquals(september.numberOfDaysIn(y2k), 30);
    assertEquals(september.monthNumber(), 9);
}
// los test para los meses de 31 días y para febrero
// con 28 y 29 días siguen el mismo patrón
```

También podríamos haber optado por hacer que en el caso de febrero se arroje una excepción y no tener que pasar un año para conocer la cantidad de días que tiene el mes. Podemos darnos cuenta de que el problema es que obligamos a todos los clientes a tener que manejar una excepción, lo cual es engorroso.

> EL OBJETO MES DE FEBRERO NO ES LO MISMO QUE EL MES DE FEBRERO DEL AÑO 2011

Recordemos el test del año. ¿Por qué ahí pudimos saber cuántos días tenia febrero? Porque ese objeto en realidad representa un mes en un determinado año y por lo tanto sabe cuántos días tiene. Es decir, sabemos que los meses de un año determinado son distintos de los meses en general (lo cual es completamente obvio).

Ahora ya podemos proceder a modelar una fecha en particular. Una fecha la podemos ver como la combinación de un año, un mes y un día (que representaremos con un número).

```
@Test public void testDateCreation() {
    final Year year1810 = Year.number(1810);
    final Month may = Month.may();
    final Date date = new Date(year1810, may, 25);

    assertEquals(date.year(), year1810);
    assertEquals(date.month(), may);
    assertEquals(date.day(), 25);
}
```

Listo, esta es nuestra batería inicial de test que nos servirán para construir nuestra solución al problema de las fechas.

Procedamos a modelar el tiempo. ¿Qué es el tiempo? Nosotros demarcamos el transcurso del tiempo utilizando horas, minutos y segundos (y también podemos seguir agregando precisión según necesitemos, pero dejemos este problema para otro momento). Las horas pueden ser de cero a veinticuatro o de cero a doce si usamos **AM** y **PM**. Los minutos y segundos van de cero a cincuenta y nueve. Cada vez que una de las partes se pasa de su límite, la siguiente parte avanza en una unidad. En particular cuando tengamos fecha y hora juntas, al

pasarse la hora de su límite, el día tiene que avanzar. Simplificaremos nuestra tarea pensando en un modelo sencillo donde el tiempo está medido en horas, minutos y segundos. Escribamos algunos test.

```
@Test public void testCreationSuccess() {
    final Time teaTime = Time.hoursMinutesSeconds(17, 0, 0);

    assertEquals(teaTime.hours(), 17);
    assertEquals(teaTime.minutes(), 0);
    assertEquals(teaTime.seconds(), 0);
}
```

Y por supuesto los test para los casos inválidos de creación.

```
@Test(expected=IllegalArgumentException.class)
public void testCreationFail_Hours() {
    Time.hoursMinutesSeconds(30, 0, 0);
}

@Test(expected=IllegalArgumentException.class)
public void testCreationFail_Minutes() {
    Time.hoursMinutesSeconds(10, 60, 0);
}

@Test(expected=IllegalArgumentException.class)
public void testCreationFail_Seconds() {
    Time.hoursMinutesSeconds(10, 0, -4);
}
```

Con esto ya tenemos una base para poder trabajar con fechas y horas, ahora nos resta agregar los métodos que necesitemos y otros conceptos que puedan surgir (como zonas horarias). Dejamos para el lector de esta obra seguir el ejercicio y finalmente obtener un objeto que represente un instante (fecha y hora). Como ejercicio adicional, imagínese como incluir en este diseño mas precisión en la hora. Pudiendo elegir hasta donde necesita: milisegundos, nanosegundos, etc.

I/O

En esta sección veremos las clases que nos permitirán acceder al **filesystem** de nuestra máquina, leer y escribir archivos. Estas clases se basan en la composición de **stream**s (corrientes de datos) tanto para leer como también para escribir. Recordemos que cada tipo de **stream** ofrece un nivel de abstracción distinto (objetos, tipos primitivos, bytes, etcétera) y distintos medios (de la red, de un archivo, etcétera). Luego existen los **writer**s y los **reader**s, que se encargan de ofrecer una visión de **stream** de caracteres puramente.

java.io.InputStream y su familia

La clase **InputStream** es la clase padre de todos los **stream**s de lectura. Define principalmente métodos para lectura de bytes (**read**) los cuales permiten leer un byte, o muchos, y copiarlos en un **array**. En caso de error, los métodos devuelven **-1** como indicador de que no hay más información para leer, signo de un diseño antiguo y no orientado a objetos. Todos arrojan una **IOExcepcion** si hay algún error.

```
// devuelve un byte o -1 si no hay más datos
int read()
// devuelven -1 si no hay más datos
int read(byte [] datos)
int read(byte [] datos, int desde, int cuantos)
```

Por ejemplo, si queremos leer un archivo de nuestra computadora, tenemos que utilizar un tipo de **stream** para acceder al **filesystem**.



DATE4J

Una librería simple que podemos utilizar como reemplazo de Date es Date4J, **www.date4j.net**. Ofrece fechas inmutables y una API muy sencilla para manipular fechas y horas. Principalmente enfocada en la interacción entre Java y las bases de datos en el tema del almacenamiento de fechas. Vale la pena darle una mirada ya que es mucho más pequeña que Joda Time.

```
FileInputStream archivo = new FileInputStream('autoexec.bat');
byte [] datos = new byte[250];
try {
    // lee hasta los primeros 250 bytes del archivo
    archivo.read(datos);
} finally {
    // NUNCA olvidarse de cerrar los streams
    archivo.close();
}
```

Es importante notar que siempre hay que cerrar los **stream**s; si no lo hacemos, nos enfrentamos con errores extraños en la aplicación.

java.io.OutputStream y su familia

No es una sorpresa que esta familia de clases se encargue de los **stream** de escritura. Opuestamente a los **InputStream**s, este tipo de objetos define métodos para escribir (**write**) bytes a algún destino. Igualmente, todos los métodos de escritura, pueden arrojar excepciones del tipo **IOException** en caso de error.

```
void write(byte byte)
void write(byte [] bytes)
void write(byte [] bytes, int desde, int cuantos)
// fuerza la escritura de todos los datos pendientes
void flush()
```



JAVA NIO

Debemos tener en cuenta que además de la forma tradicional que tiene Java para manejar entrada y salida de datos, existe lo que se conoce como **New I/O**. **NIO** define nuevas clases e interfaces para manejar de forma eficiente y asincrónica las lecturas y escrituras de datos. Si bien no es muy conocida y utilizada, existen productos que la usan como el servidor **Jetty**.

De forma similar al ejemplo de la escritura, tratemos ahora de escribir un archivo, como vemos en el código siguiente.

```
FileOutputStream salida = new FileOutputStream("saludos.txt");
try {
    salida.write("hola a todos!".getBytes());
salida.flush();
} finally {
    salida.close();
}
```

Recordar que hay que cerrar los **stream**, también los de salida.

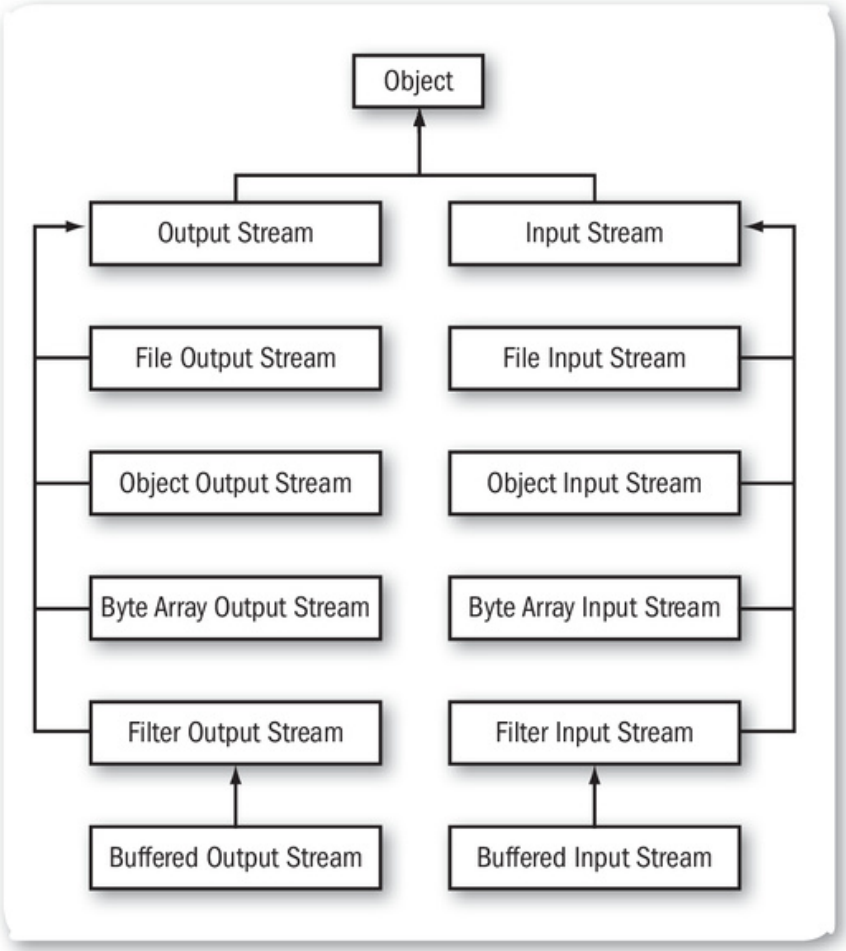


Figura 4. Aquí vemos la jerarquía de los **`stream`**s de lectura y escritura. Notar la simetría en las jerarquías.

java.io.Reader y java.io.Writer

Esta familia de clases se asemeja a los **stream**s de lectura y escritura pero se focaliza en caracteres y **String**s en vez de hacerlo en bytes. El protocolo definido por esta familia es similar al de **InputStream** y **OuputStream** pero con la adición de algunos métodos.

Veamos un segmento del protocolo definido por **Reader**.

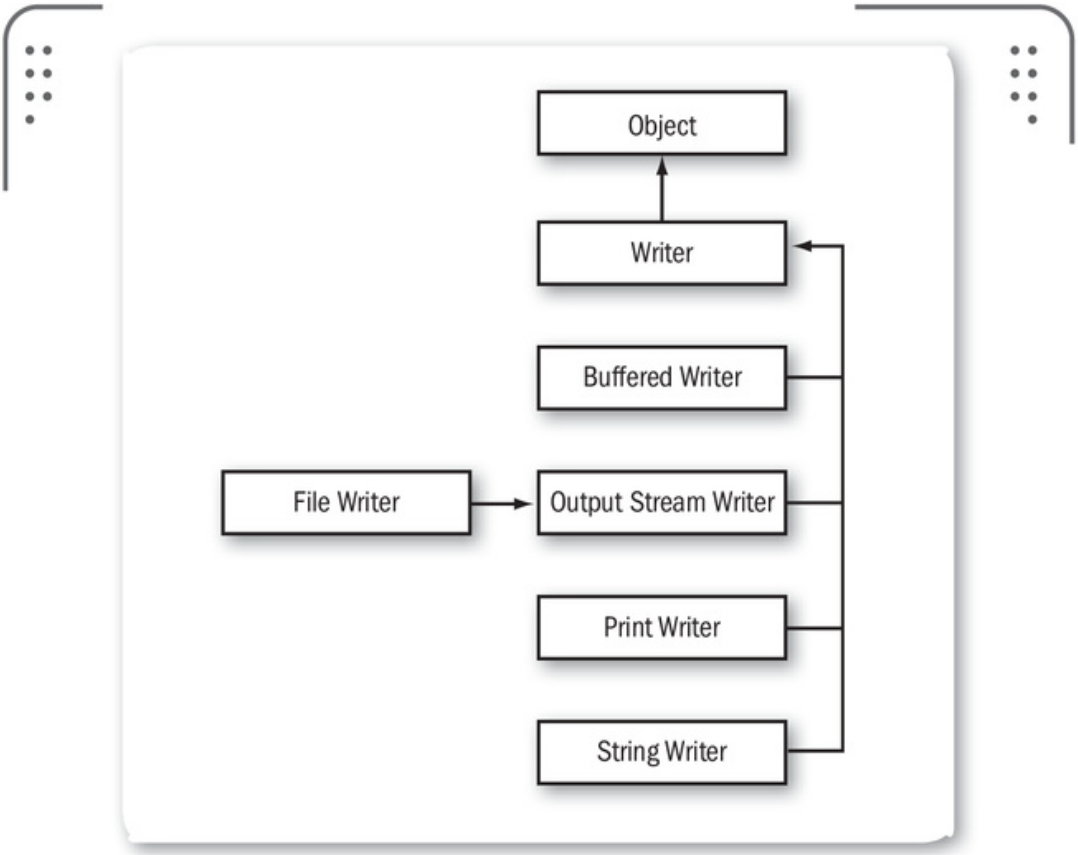


Figura 5. Esta imagen nos presenta un diagrama de la jerarquía que corresponde a los **Reader**.

AUTOCLOSEABLE EN JAVA 7

Es necesario tener en cuenta que en la futura versión de Java se prevé el agregado de un nuevo uso al **try** para que permita operar con **stream**s sin preocuparnos de cerrarlo correctamente. Esta nueva forma se conoce como **try-with-resource** y es sencillamente una forma más fácil de escribir el mismo código que veníamos escribiendo (**try** y luego un **finally** para cerrar el **stream**).

```
// devuelve un carácter o -1 si no hay más datos
int read()
// devuelven -1 si no hay más datos
int read(char [] caracteres)
int read(char [] caracteres, int desde, int cuantos)
int read(CharBuffer contenedor)
```

Y ahora presentamos algunos de los métodos declarados por **Writer**.

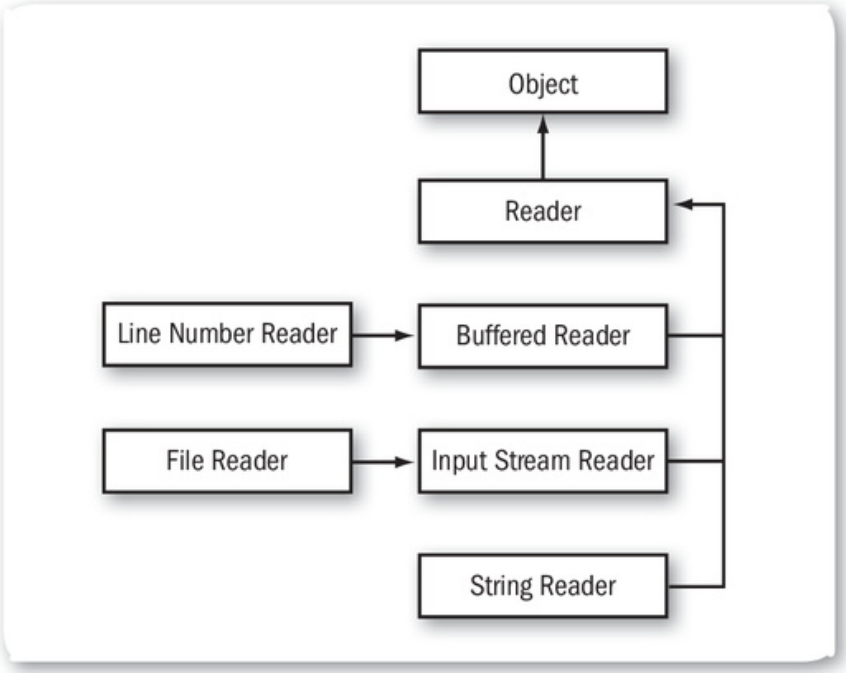


Figura 6. En esta imagen podemos apreciar el diagrama de la jerarquía correspondiente a **`Writer`**.



COMPOSICIÓN DE STREAM

Una de las mejores características de diseño de la **API** de **I/O** es que se pueden ir componiendo distintos objetos para que podamos obtener distintas funcionalidades. Un ejemplo muy común es componer un **stream** de lectura con un **BufferedInputStream** para obtener una mejor performance y luego con un **ObjectInputStream** para leer datos concretos (números, **String**s y objetos).

```
Writer append(char caracter)
// recordar que String hereda de CharSequence
Writer append(CharSequence secuenciaDeCaracteres)
// ¿Extraño no? ¿Por qué no acepta un carácter?
void write(int caracter)
void write(String texto)
void write(char [] caracteres)
void close()
void flush()
```

Tengamos en cuenta que aquí debemos tener las mismas consideraciones que cuando trabajamos con **stream**.



RESUMEN

Esta fue una breve introducción a las clases más básicas que se ofrecen en la librería estándar de Java. Estas son las clases e interfaces que todo programador no puede desconocer. Las colecciones son parte fundamental de todo programa. Por lo tanto hay que conocer bien cuáles son las capacidades de cada una de las implementaciones para poder seleccionar la más adecuada a la tarea. Las de manipular fechas y horas también son importantes, ya que un mal uso puede llevar a errores con los husos horarios que son difíciles de encontrar. Por último, conocer cómo transmitir y recibir información, ya sea de la Red o de un archivo, es básico y merece un estudio más profundo.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Qué cuidados se debe tener al implementar equals?
2	¿Y al implementar hashCode?
3	¿Cuál es la diferencia entre StringBuilder y StringBuffer?
4	¿Qué tipo de colecciones ofrece Java?
5	¿Cuáles son las implementaciones más conocidas y usadas de cada una?
6	Imagine un uso para un Set.
7	Piense un escenario de uso para Map.
8	¿Por qué se utiliza un calendario para manipular las fechas?
9	¿Qué deficiencias tiene el diseño de Date?
10	¿Cuál es la principal ventaja en el diseño de la jerarquía de I/O?

ACTIVIDADES PRÁCTICAS

1	Agregar la interfaz Zippable, que ofrezca la funcionalidad zip.
2	Agregar la interfaz Filterable, que permite filtrar una colección dada una condición.
3	Agregar una interfaz que define la capacidad para iterar una colección y actuar sobre sus elementos (forEach).
4	Modelar operaciones sobre las horas (suma, incremento, etcétera).
5	Analizar cómo definir la idea de intervalos temporales.

*11

Anotaciones

Las anotaciones son un mecanismo para poder unificar toda esta información extra código en una sola forma estandarizada. Esto permite no solo que los programadores no tengan que inventar formas de volcar la información extra, sino que también se ahorren el tema de manipularla. Aquí aprenderemos a usarlas en nuestros proyectos.

- **¿Qué son las anotaciones? 200**
- **Definición 204**
  - Heredando anotaciones 208
- **Uso de las anotaciones 209**
- **Accediendo a las anotaciones en tiempo de ejecución 209**
  - Jerarquizando anotaciones 212
- **Distintos usos de las anotaciones 217**
  - Inyección de dependencias 219
  - Serialización 220
  - Mapeos a base de datos 222
  - Aplicaciones Web 223
- **Resumen 223**
- **Actividades 224**

¿Qué son las anotaciones?

Durante la historia y evolución de Java se inventaron muchas formas de agregar información pertinente al código pero separada de él, datos que lo complementaban. Por ejemplo, los **JavaBeans** (objetos originalmente pensados para ser editados gráficamente) requerían la definición de una clase acompañante del tipo **BeanInfo**, donde se mantenía la descripción de bean. Los famosos **EJB** o **Entrerprise Java Beans** (sí, **beans;** en Java, **beans** -granos de café- significa objetos, para aplicaciones empresariales) requerían en sus versiones anteriores varias interfaces extra para poder ser usados en las aplicaciones de servidores. Incluso las aplicaciones web Java requieren varios archivos **XML** para poder ser configuradas. También antiguamente se utilizaban comentarios en el código, que seguían cierto formato y que se valían de algunas herramientas (incluido el compilador) para funcionar en forma correcta.

LAS APLICACIONES WEB JAVA REQUIEREN ARCHIVOS XML PARA SER CONFIGURADAS

Las anotaciones son artefactos que permiten especificar información extra respecto de una clase, método o variable, directamente en el código y de forma tal que estas pueden ser accedidas como objetos por el sistema. Debemos saber que esta información que habla sobre los elementos de un programa se conoce como **metadata** (que básicamente quiere decir datos sobre los datos).

La presencia de anotaciones no afecta, inicialmente, la semántica del código. Pero una herramienta o librería puede leer esta información y actuar modificando el comportamiento de los elementos anotados (es decir, a aquellos afectados por las anotaciones). En Java las anotaciones

JAVADOC Y XDOCLET

Desde sus comienzos y antes de que aparecieran las anotaciones existían comentarios de código que servían para el mismo propósito. Estos se conocían como **xdoclet** y ofrecían una alta gama de tags para distintas funcionalidades. Se requería utilizar una herramienta extra que leía el código fuente y operaba en consecuencia a medida que iba encontrando los tags en los comentarios.

se escriben junto con los modificadores del elemento en cuestión y su formato es el símbolo @ más el nombre de la anotación; opcionalmente, se escriben entre paréntesis una serie de argumentos para la anotación.

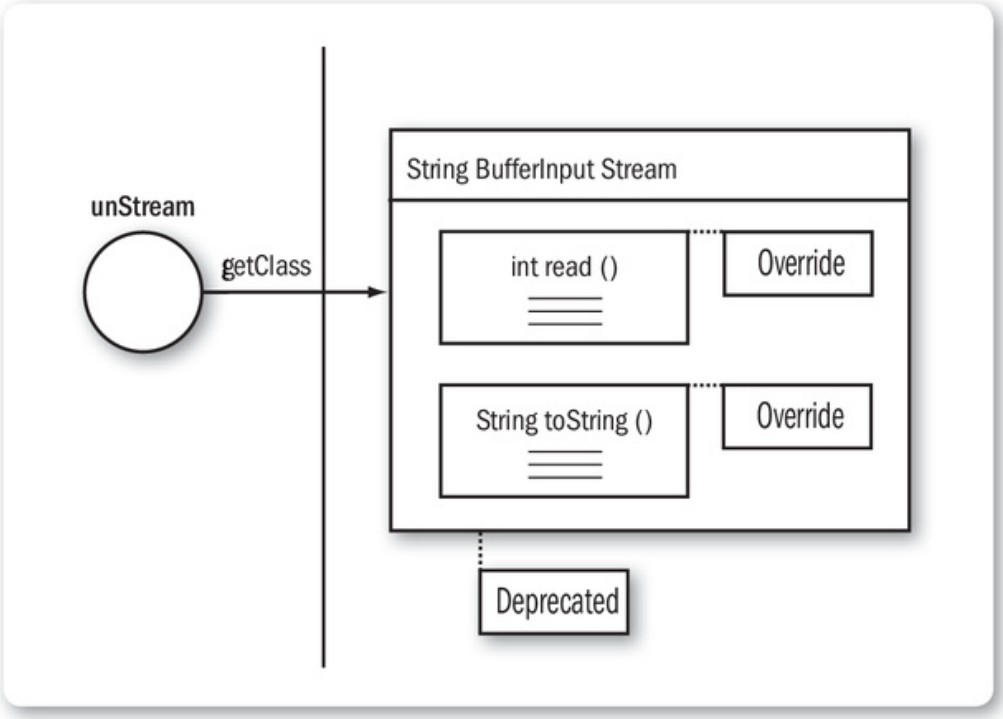


Figura 1. Las anotaciones no afectan la semántica de una clase, sino que agregan información a sus elementos.

Algunas anotaciones conocidas

Hemos estado utilizando algunas de estas anotaciones en nuestros códigos que se encuentran en los capítulos anteriores. Para continuar en esta línea repasemos algunas de ellas.

@Override

Esta anotación puede, en principio, parecer innecesaria, ya que el compilador sabe si estamos sobrescribiendo un método o no. Es conveniente utilizarla siempre por varios motivos. Por ejemplo, si queremos sobrescribir un método y nos equivocamos al momento

de escribir el nombre, el compilador nos alertará de que no estamos sobrescribiendo un método conocido. Solamente está indicada para ser utilizada en tiempos de compilación. También es una forma de asegurarnos de que estamos implementando los métodos de una interfaz correctamente. Esta anotación solo puede ser aplicada sobre métodos.

@Test

Otra anotación que utilizamos frecuentemente es **@Test**, la cual es usada en nuestros **Test Cases** para indicar cuáles métodos representan pruebas unitarias. Esta anotación puede aceptar algunos argumentos en su uso. Por ejemplo, si deseamos indicar que el test arroja una excepción, lo hacemos con el argumento **expected**.

```
@Test(expected=IOException.class)
```

También es posible indicar que la prueba debe tardar menos de una determinada cantidad de tiempo o de lo contrario falla. Lo hacemos agregando el argumento **timeout** y especificando a continuación la cantidad de milisegundos que se debe esperar.

```
@Test(timeout=1000)
// esperamos un segundo o falla
```

Es necesario entender que ambos argumentos pueden ser utilizados al mismo tiempo. La anotación **@Test** solamente puede decorar métodos que sean públicos y no devuelvan nada (**void**).



ANOTACIONES EN JUNIT

En las versiones viejas de **JUnit** se utilizaba la convención de que los métodos que representaban pruebas unitarias eran aquellos cuyos nombres empezaban con el prefijo test. Afortunadamente esto se dejó de usar, ya que un simple error de tipeo podía hacer que un test no pudiera ser corrido y, por lo tanto, que estuviéramos ante una serie de errores en nuestro código.

@Deprecate

Esta anotación es utilizada para indicar que un elemento (clase, interfaz, método, etcétera) no debe utilizarse más y que es posible que en futuras versiones sea removido por completo. Generalmente se usa cuando un diseño es actualizado y se conservan los elementos antiguos para mantener la compatibilidad hacia atrás.

Anteriormente esta funcionalidad era ofrecida por un comentario que contenía el texto **@deprecated**. Tengamos en cuenta que see espera que el compilador alerte al programador de tales usos. Por otro lado, es importante señalar que **@Deprecated** puede ser utilizada para decorar cualquier elemento.

@SuppressWarnings

Sirve para apuntarle al compilador que debe dejar de indicar alarmas del tipo especificado sobre un elemento determinado. Los tipos de alarma los especificamos utilizando su nombre (en texto) y podemos indicar más de un tipo de alarma al mismo tiempo (con un array). En la siguiente tabla vemos las alarmas más comunes.

ALARMAS

▼ ALL	▼ SUPRIME TODAS LAS ALARMAS
deprecation	Suprime las alarmas de uso de código deprecado.
unchecked	Anula las alarmas de uso de llamadas o casteos no verificados (en tipos).
serial	Deshabilita las alarmas de las definiciones de tipos **Serializables** sin un **serialVersionUID**.
rawtypes	Deshabilita las alarmas relacionadas con el uso de tipos genéricos sin especificar los tipos.
finally	Anula las alarmas de los **returns** dentro de los **finally** (ya que ocultan el **return** del **try**).
unused	Cancela las alarmas relacionadas con variables no utilizadas.

Tabla 1. Alarmas que es posible especificar.

Definición

La definición de una anotación no es muy distinta de la definición de cualquier interfaz. La diferencia clave es la utilización del símbolo @ antes de la palabra **interface**.

```
public @interface MiAnotacion {
   ...
}
```

Ahora, los argumentos que se le pueden pasar a una anotación se definen como si fueran métodos de la interfaz pero con algunas restricciones. No pueden tener parámetros o indicar que arrojan una excepción (utilizando **throws**). Los tipos de retorno están restringidos a los tipos de datos primitivos (**byte**, **char**, **short**, **int**, **long**, **float** y **double**), cadenas de caracteres (**String**), clases (**Class**), otras anotaciones y **arrays** de los tipos anteriores. Así mismo, los métodos pueden definir un valor por defecto, declarándolo con la palabra **default** seguida del literal apropiado. Notamos que solamente podemos indicar literales como valores por defecto y los tipos son los permitidos por las anotaciones.

```
public @interface Test {
    Class<? Extends Throwable> expected()
        default None.class;
    // declaramos el valor por default como 0 (nada)
    long timeout() default 0L;
}
```



NOMBRE DE LOS ARGUMENTOS

Ya vimos algunos consejos sobre los argumentos de las anotaciones, pero ellos no son todo. Siempre pensemos en cómo se usaran las anotaciones y de qué forma el cliente utilizará los argumentos. Debe ser bien claro cuál es el propósito de cada uno y cuáles son los valores permitidos. Tratemos de evitar los **String**s para otra cosa que no sea texto libre (no identificadores), a menos que sea necesario.

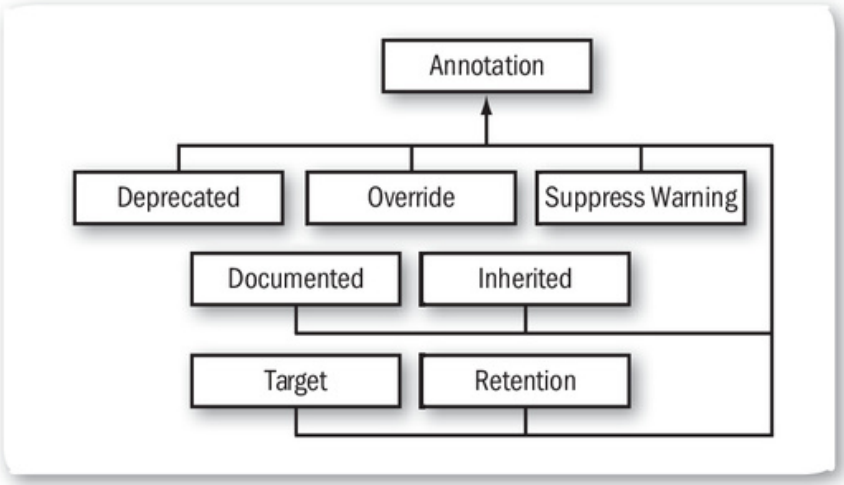


Figura 2. Jerarquía de las anotaciones en Java. Podemos notar que está formada por interfaces.

Debemos tener en cuenta que cuando utilizamos un array, es posible indicar solamente un elemento y automáticamente será tomado como un array. Además, si nuestro parámetro se llama **value**, entonces al momento de utilizarlo no es necesario especificar su nombre y podemos pasar directamente el valor que queremos.

```
public @interface SuppressWarnings {
    String [] value();
}
// no tenemos que especificar value ni el array
@SuppressWarnings("rawtypes")
Map a = new HashMap();
// en este caso sí los ponemos como array porque son más de uno
@SuppressWarnings({ "rawtypes", "unchecked" })
Map<String,String> b = (Map<String,String>) new HashMap();
```

Cuando utilizamos las anotaciones, los valores que usamos como argumentos deben ser literales (ya que son evaluados en tiempo de compilación). Las anotaciones que no tienen definido ningún método se conocen como **markers** (marcadores), ya que solo dan información por su presencia (marcan el elemento decorado). Por su parte, las

anotaciones que solamente tienen un argumento deberían llamarlo **value**, así no es necesario especificarlo cada vez que se las quiere utilizar.

Las anotaciones no pueden extender ninguna interfaz y tampoco pueden ser implementadas. Por lo tanto, una anotación no puede heredar de otra y tampoco forma jerarquías de tipos.

Al momento de definir las anotaciones, debemos indicar sobre qué elementos pueden ser aplicadas y cuál es el alcance que tienen. Para definir los elementos objetivos de la anotación, decoraremos la declaración de esta con la anotación **@Target**. Ella permite especificar, mediante un array de enumeraciones del tipo **ElementType**, los objetivos. **ElementType** define los siguientes elementos: **TYPE** (clases e interfaces), **CONSTRUCTOR** (constructores), **METHOD** (métodos), **FIELD** (atributos), **LOCAL_VARIABLE** (variable local), **PARAMETER** (parámetro de un método), **ANNOTATION_TYPE** (declaración de una anotación, para definir meta anotaciones) y **PACKAGE** (declaración de un paquete). También para definir el alcance de la anotación lo hacemos decorándola con **@Retention**, que especifica si la anotación es solamente para tiempo de compilación (**SOURCE**), si debe estar en el binario de la clase pero no necesariamente en tiempo de ejecución (**CLASS**) y si debe ser mantenida, incluso en tiempo de ejecución, para poder ser inspeccionada (**RUNTIME**). Esta enumeración corresponde al tipo **RetentionPolicy**.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
   ...
}
```



ANOTACIONES EN TIEMPO DE COMPILACIÓN

La mayoría de las anotaciones caseras se utilizan en tiempo de ejecución, pero también existe la opción de operar con ellas en tiempo de compilación. Java provee una herramienta llamada **APT** (**Annotation Processing Tool**), que permite enganchar nuestro código y dejarnos generar código, elevar alarmas o errores de compilación que dependen de las anotaciones encontradas en los códigos fuentes.

@Retention y **@Target** son llamadas meta anotaciones ya que son anotaciones que decoran otras anotaciones, es decir, se trata de anotaciones sobre anotaciones.

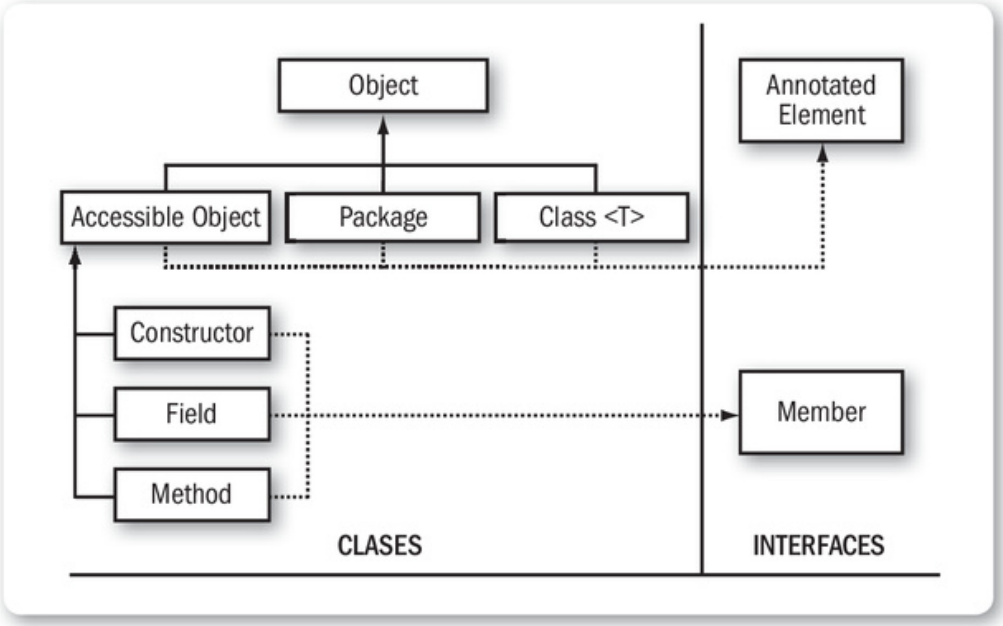


Figura 3. En esta imagen podemos ver la jerarquía de los elementos que pueden ser decorados con anotaciones.

Eclipse ofrece una opción para crear rápidamente una anotación básica. Para realizar esta tarea, debemos seleccionar **File** o presionar el botón **New** en la barra de herramientas, elegimos la carpeta de los archivos fuentes e ingresamos el paquete donde deseamos crearla. Escribimos el nombre de la anotación y presionamos **Finish**.

SPRING FRAMEWORK

En el sitio web que se encuentra en la dirección **www.springsource.org** existe este famoso framework cuyo principal motivo es la inyección de dependencias. Spring fue el primer framework que apareció para este propósito y marcó una tendencia. Actualmente **Spring** ofrece una enorme cantidad de otras funcionalidades que lo enriquecen, desde el desarrollo de aplicaciones web hasta el acceso a bases de datos, o funciones de mensajería y seguridad, entre otros.

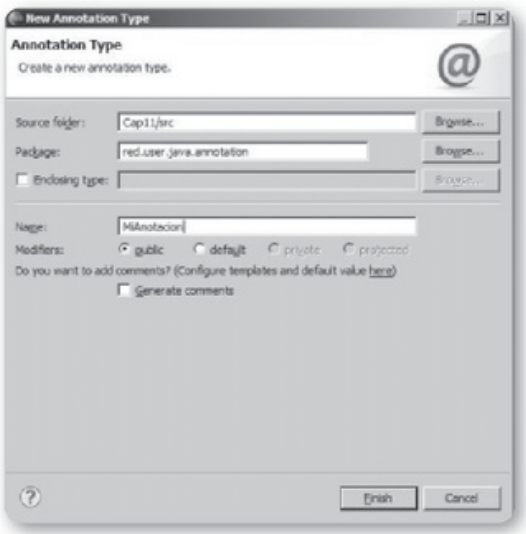


Figura 4. En esta imagen podemos ver la ventana de creación de anotaciones en Eclipse.

Heredando anotaciones

En principio cuando anotamos una clase, no se considera que las subclases estén anotadas también, o sea, que la anotación afecte a las clases hijas. Este es por lo general el comportamiento deseado. Pero si estamos seguros de que lo que queremos es que se propaguen las anotaciones en la jerarquía, debemos especificarlo. Para especificarlo tenemos que, al momento de definir la anotación que nos interesa que se herede, marcarla con la meta anotación **@Inherited**. Esta permite que la anotación decorada con ella pueda ser heredada y transmitida a las clases hijas. Igualmente, heredar la anotación no significa que la clase hija declara el uso de la anotación, son dos cosas bien distintas.

Nos conviene mencionar que solamente cuentan las anotaciones heredadas desde una superclase. No podemos heredar anotaciones que estén definidas en una interfaz que implementemos.



AYUDA DEL IDE

Una vez más la asistencia del **Eclipse IDE** es invaluable. Eclipse nos ayuda a completar los nombres de las anotaciones y, lo más importante, nos muestra cuáles son los argumentos que estas soportan y de qué tipo son. Recordemos que para invocar el **autocomplete**, debemos presionar la tecla **CTRL + BARRA ESPACIADORA** sobre la anotación y en el lugar donde van los argumentos.

Uso de las anotaciones

El uso de las anotaciones es sencillo y ya lo hemos experimentado. Las anotaciones se utilizan de la misma forma que se utilizan los modificadores, van siempre delante del elemento. Por convención se espera que estas encabecen la lista de modificadores o que estén en la línea inmediatamente superior a la declaración del elemento en cuestión.

```
@Test public void testSomething() {
    ...
}

@Test(expected=RuntimeException.class)
public void testSomeErrorCase() {
    ...
}
```

Si tratamos de utilizar una anotación en un elemento que no corresponde, obtendremos un error de parte del compilador.

Accediendo a las anotaciones en tiempo de ejecución

Ahora vamos a ver cómo podemos, desde nuestro código Java, acceder a la información declarada en las anotaciones de los objetos de nuestro sistema. Tratando de seguir la filosofía de que todo es un objeto, Java provee ciertas clases que modelan a las clases, los métodos y demás elementos, como constructores, atributos, parámetros y paquetes. Nosotros ya conocemos cómo acceder a la clase de un objeto en particular, le enviamos el mensaje **getClass()**. También podemos obtener el objeto clase si utilizamos un literal de clase como **String.class**. Ahora que tenemos acceso al objeto que representa la clase, le podemos

pedir los demás elementos. Las clases que representan a los distintos elementos implementan la interfaz **AnnotatedElement**, que ofrece un protocolo para obtener las anotaciones relacionadas con el elemento.

```
// protocolo de AnnotatedElement

// devuelve las anotaciones de un tipo determinado
<T extends Annotation> getAnnotation(Class<T> annontationClass);

// devuelve todas las anotaciones del elemento (declaradas y heredadas)
Annotation[] getAnnotations();

// devuelve todas las anotaciones directamente declaradas en el elemento
Annotation[] getDeclaredAnnotations();

// responde si el elemento esta anotado con un tipo de anotación en particular
boolean isAnnotationPresent(Class<? extends Annotation> annotationClass);
```

El lector alerta habrá notado el uso del tipo **Annotation** en el protocolo anterior. Esta interfaz es la representación del uso de una anotación en un elemento en tiempo de ejecución. Recordemos que las anotaciones son interfaz, no tienen implementación concreta. Entonces, ¿dónde está definida la implementación? La realidad es que el soporte de tiempo de ejecución crea una clase oculta que implementa la interfaz de nuestra anotación y que implementa **Annotation** (al implementar nuestra anotación), de ahí que el tipo usado sea ese. Veamos un ejemplo de cómo obtener la **metadata** de una clase.

```
public class AnnotationUnitTests {

 @Test(timeout=150)
 public void testXXX() {
// este método va a ser utilizado en el test siguiente
 }

@Test
```

```
 public void testTestAnnotation() {

// consiguimos la clase y le pedimos el método
  Method method =
this.getClass().getMethod("testXXX");

// pedimos la anotación
  Test test = method.getAnnotation(Test.class);

// ahora le pedimos a la anotación el timeout
  long timeout = test.timeout();

// y ahora el error esperado
  Class<? extends Throwable> expected =
test.expected();

// verificamos que el timeout sea el correcto
  assertEquals(timeout, 150);

// y que no se espere ningún error
  assertEquals(None.class, expected);

 }

}
```

En el código anterior podemos observar cómo consultamos al objeto de nuestro test case por su clase, y luego a esta por un método del cual



REFLEXIÓN

El termino reflexión se utiliza para indicar la capacidad de un sistema de mirarse a sí mismo, de evaluarse y, eventualmente, de modificarse. Específicamente esto sucede cuando los meta objetos del sistema pueden ser usados como objetos comunes y corrientes. En Java tenemos una clase que representa a las clases, otra que representa a los métodos y así con cada elemento de nuestro programa. Cuando pedimos la clase de un objeto, estamos pasando al lado de los meta objetos.

sabemos su nombre. Cuando obtenemos el método que nos interesa le pedimos que nos devuelva la anotación del tipo **@Test**. Ahora tenemos en nuestras manos la instancia, un objeto, que representa la información que pusimos en nuestro código. Obtenemos entonces del objeto anotación el **timeout** especificado y el error esperado (que en este caso es ninguno), y validamos que lo que decimos sea correcto.

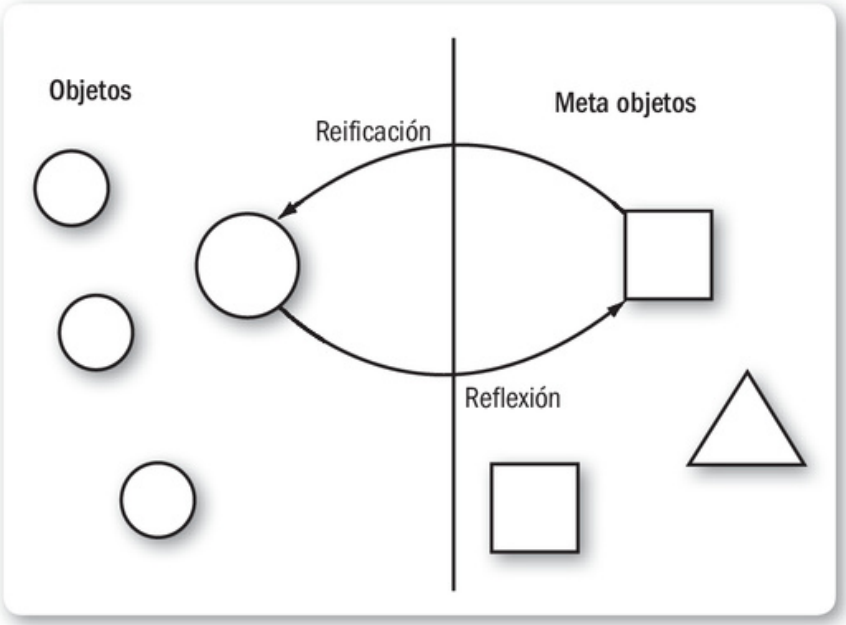


Figura 5. Visualización de objetos según pertenezcan a un nivel meta o no. El pase del nivel de dominio al nivel meta es mediante un mensaje, como **`getClass`**.

Jerarquizando anotaciones

Las anotaciones por más que sean, en definitiva, interfaces, no se permite que extiendan ninguna otra anotación (ni interfaz). Esta es una limitación que solamente se hace evidente cuando queremos agrupar o jerarquizar distintas anotaciones. Supongamos que queremos crear un conjunto de anotaciones para imponer restricciones en los argumentos de los métodos (que no sea nulo, que sea positivo, que sea una lista no vacía, etcétera). Cuando queramos acceder a estas anotaciones deberíamos saber cuáles estamos buscando. Podemos saberlo por enumeración, conociendo todas las anotaciones de restricción que

hayamos creado. Esta aproximación a la solución tiene el problema que al agregar nuevas anotaciones debemos cambiar esta búsqueda o idear algún mecanismo para notificar de la existencia de una nueva restricción. Otra forma de encarar esto es catalogando las anotaciones. No podemos hacerlo usando herencia, como ya dijimos, pero podemos decorarlas con otras anotaciones. De esta forma, podemos crear una meta anotación que identifique las restricciones y luego, al buscarlas, solo debemos asegurarnos de preguntar si están catalogadas como restricción o no. Así las nuevas restricciones que creemos (nosotros u otros) estarán naturalmente incluidas sin trabajo extra. Veamos como sería tal meta anotación.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Restriccion {

}
```

Y a continuación cómo se decoraría la restricción para que un argumento de un método no sea nulo.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
@Restriccion
public @interface NoNulo {

}
```



¿EXTENDER LAS ANOTACIONES?

Java decidió no permitir extender las anotaciones mediante herencia debido a que, según el comité que aprueba los cambios en Java, haría más difícil la escritura de herramientas específicas para manipular las anotaciones. También aseguran que habría que mantener un sistema de tipos paralelo solamente para las anotaciones, que agregaría más complejidad tanto al compilar como a la máquina virtual.

Al buscar las anotaciones debemos preguntar por su categoría. Por ejemplo, una forma de hacerlo sería esta.

```
// supongamos que annotation es una anotación de un argumento de un método
que nos interesa
if(annotation.annotationType()
        sAnnotationPresent(Restriccion.class)) {
  // hacemos lo que tengamos que hacer
} else {
  // no hacemos nada con esta anotación
}
```

Esta forma de trabajo, combinada con lo que aprenderemos en la sección siguiente nos permitirá programar código altamente flexible y extensible para lidiar con las anotaciones.

Comportamiento en las anotaciones

Como hemos visto en la sección anterior, las anotaciones son, en definitiva, interfaces que no podemos implementar directamente. Entonces, ¿cómo podemos agregarles comportamiento y que no sean solamente contenedoras de información? Debemos crear alguna otra clase que contenga el comportamiento asociado a la anotación. Para esto lo mejor es mantener dicha anotación y dicha clase lo más cercanas y relacionadas posible. Si recordamos el capítulo sobre los distintos tipos de clases, seguramente recordaremos las clases internas, públicas y estáticas. Al ser interfaces las anotaciones permiten los mismos elementos que ellas, por lo tanto podemos crearle clases internas que sean las que operen sobre ella y le provean cierto comportamiento manteniendo la cohesión, ya que está todo definido en conjunto. Veamos un ejemplo. Supongamos que necesitamos una anotación para indicar que queremos memorizar los resultados de un método que es lento, así, la próxima vez que lo utilicemos con los mismos resultados, la respuesta sea instantánea (esto se conoce como **memoization**). Nuestro caso de prueba será el método para obtener el factorial de un número entero. Recordemos que el factorial (**!**) de un número entero **n** está definido como **1 si es 0** ó **como n x (n - 1)!** en otros casos.

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)\,! \times n & \text{if } n > 0. \end{cases} \qquad n! = \prod_{k=1}^{n} k$$

Figura 10. En esta imagen podemos ver dos definiciones que corresponden a la función factorial.

```
public class Entero {
  ...
  @Memoize
  public Entero factorial() {
    if(this.esCero()) {
      return uno();
    }
    return this.multiplicadorPor(this.anterior().factorial());
  }
  ...
}
```

Hasta aquí hemos podido observar que ya hemos anotado el método **factorial** para ser memorizado. De esta forma, cuando queramos el factorial de 4 (4!), la primera vez se calcularán los factoriales de 1, de 2, de 3 y de 4 y quedarán memorizados. La próxima vez que pida el factorial de alguno de ellos, este será obtenido sin recalcular nada y sin invocar el factorial que corresponde a los otros números.

SOBRE MEMOIZATION Y CACHING

Recordar resultados pasados para no tener que realizar el trabajo de obtenerlos nuevamente es una técnica ampliamente utilizada. Desde los servidores web, que memorizan páginas completas, hasta los números factoriales, como en nuestro ejemplo. Hay que mencionar que no es fácil implementar correctamente estos mecanismos, ya que debemos manejar cuánta memoria estamos usando y su liberación.

Ahora nos toca crear la anotación, que en principio solamente es un **marker**, así que no le definimos ninguna propiedad. Obviamente nuestra anotación debe tener alcance de tiempo de ejecución, ya que nos interesa modificar el comportamiento mientras se ejecuta la aplicación y solamente queremos recordar métodos particulares.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Memoize {

}
```

Luego necesitamos codificar todo lo necesario para recordar que cuando se llama a determinado método de determinada clase con una lista de parámetros determinados, debemos devolver el valor recordado o invocar el método, recordar el resultado para después y devolverlo. Todo este código debería ser cercano a **@Memoize**, por lo tanto vamos a ponerlo en una clase interna estática y pública.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Memoize {
  public static class Apply {
    ...

    public static <T> T to(T target) {
      // acá va el código necesario para devolver un objeto con el comportamiento
de memorización
    }
    ...
  }
}
```

Una vez que tengamos esto podemos usarlo para crear instancias que puedan recordar los mensajes enviados a ellas y devolver la respuesta instantáneamente en vez de volver a ejecutar el método

asociado. Esta es una práctica muy común (también se la conoce como **caching**) que es usada para mejorar la velocidad de código crítico. Notemos que el hecho de que se recuerden los resultados es totalmente ortogonal a cómo es el objeto al que queremos aplicarle esta funcionalidad y cuál es su comportamiento. No programamos este tipo de funcionalidad directamente en cada método que queremos acelerar, sino que lo hacemos una vez y luego lo aplicamos a cada caso. Tanto la anotación como el comportamiento asociado a ella se encuentran definidos juntos, con lo cual no tenemos que adivinar cómo se utilizan y, de estar separados, dónde se encuentran.

Distintos usos de las anotaciones

Hay infinidad de aplicaciones, librerías y **frameworks** que hacen uso de las anotaciones. Ya sea para proveer cierta funcionalidad o para ampliar una funcionalidad base, los usos de las anotaciones son vastos.

Validaciones

Java propone un especificación donde aparecen varias anotaciones para validar objetos, tanto los atributos de un objeto como los parámetros de un método. Algunas de las anotaciones definidas permiten declarar restricciones sobre las referencias y los objetos, tales como no ser nulas, estar entre cierto rango número, tener una determinada longitud o si una fecha debe ser en el futuro o en el pasado.



¿QUÉ ES UN FRAMEWORK?

Un **framework** es un conjunto de elementos reutilizables que conforman un sistema con un propósito determinado. Tal sistema está incompleto en el sentido que tiene "huecos" los cuales completaremos con nuestro código que especializará el comportamiento general a nuestras necesidades. La característica principal, frente a una librería, es que el código del **framework** tiene el control total, no nosotros.

También permiten declarar precondiciones, poscondiciones e invariantes de métodos y clases. Esta práctica es conocida como diseño por contrato. Además de la especificación, existen varias librerías que, dadas esas anotaciones de una forma u otra, realizan estas validaciones.

```
class Persona {
@NotNull @Length(min=3)
        private String nombre;
        @Min(0) private int edad;

        ...

        boolean esHermanoDe(@NotNull @Valid otraPersona)
        {
        ...
        }

        ...
}

// al usar esHermanoDe con null debería arrojar una excepción
juan.esHermanoDe(null)
```

En general las restricciones y validaciones deberían estar encapsuladas en los objetos correctamente modelados. Por ejemplo, el nombre no debería ser tan solo un **String**, sino un objeto nombre propiamente dicho y dentro de él debería contener las validaciones necesarias para que las instancias sean válidas.



OVAL

Oval (**http://oval.sourceforge.net**) es un proyecto que está dedicado a la creación de anotaciones para la validación de nuestras clases y métodos. Define una amplia variedad de anotaciones de restricciones sobre los atributos, parámetros y resultados. Permite definir nuestros propios validadores y también la posibilidad de escribir código validador directamente en las anotaciones.

Inyección de dependencias

Los objetos dependen de otros objetos con los cuales colaboran para realizar un objetivo específico. Un problema de esto es cómo enlazar los objetos entre ellos. Hay casos en que los colaboradores de un objeto son externos a él, donde son pasados en el momento de la construcción o mediante un método **setter**. Otros casos son internos al objeto como, por ejemplo, una colección para mantener ciertos otros objetos (externos o no). Esta colección seguramente será creada por el mismo objeto pero, en los otros casos, ¿quién se encarga de inicializar los objetos dependencias y de pasárselos al objeto que los necesita? Nosotros mismos podemos escribir este código que pega unos objetos con otros, pero existen varios proyectos que ya solucionaron este problema por nosotros. Estos proyectos utilizan anotaciones para configurar cómo debe realizarse la inicialización de cada objeto y de sus dependencias.

> UN PROBLEMA COMÚN ES CÓMO ENLAZAR LOS OBJETOS ENTRE ELLOS

```
public class VideoClub {
  @Autowired
  private CatalogoDePeliculas catalogo;

  ...
}
```

En el ejemplo, el catálogo de películas será pasado automáticamente al objeto del videoclub en el momento de creación.



GUICE

Este framework de inyección de dependencias creado por el gigante **Google** permite la rápida y sencilla configuración de nuestros objetos. **Guice** se encarga de manejar todo el código que pega a los objetos entre sí, haciéndose cargo de su creación, configuración y cuidado. La idea es dejar de encargarse uno mismo de crear las instancias de los colaboradores para delegar esta responsabilidad a otra parte.

Serialización

Se dice que un objeto se serializa cuando es transferido fuera de la máquina virtual, ya sea a un archivo, a otra máquina virtual por la red o a un formato distinto, como un archivo **XML**. Si bien Java ofrece mecanismos para serializar semiautomáticamente objetos, lo hace de forma binaria. Hoy en día, sobretodo en la Web, se utilizan formatos basados en texto como **XML** o **JSON**. Por ejemplo, en el caso de **XML**, la traducción entre objetos y **XML** es un tanto ambigua, algunos atributos pueden ser elementos en el **XML**. Para poder configurar tal traducción muchas herramientas hacen uso de las anotaciones.

```
@Root
public class Agenda {
    ...
    @ElementList
    private List<Contacto> contactos;
    ...
}

@Root
public class Contacto {
    ...
    @Attribute
    private String nombre;

    @Element
    private Direccion direccion;

    private List<String> telefonos;
    ...
}

@Root
public class Direccion {
    ...
    @Element
   private String calle;
```

```
    @Element
    private String numero;
    ...
}
```

Si ahora queremos serializar un objeto del tipo **Agenda** obtendremos el siguiente **XML**.

```
<agenda>
 <contactos>
  <contacto nombre="Juan Perez">
   <direccion>
    <calle>Lavalle</calle>
    <numero>123</numero>
   </direccion>
   <telefono>134-23456</telefono>
  </contacto>
  <contacto nombre="Armando Tranvias">
   <direccion>
    <calle>De las ores</calle>
    <numero>56</numero>
   </direccion>
   <telefono>456-4343</telefono>
   <telefono>456-4344</telefono>
  </contacto>
 </contactos>
</agenda>
```



SIMPLE

Simple es una librería altamente útil para serialización y configuración para **XML**. Con un mínimo esfuerzo, reduce el código y errores y permite la creación de sistemas que utilicen **XML** para la comunicación y otras funciones. Su principal característica es que ofrece la posibilidad de serializar en su totalidad objetos de forma automática, incluidas las referencias a otros objetos.

Mapeos a base de datos

Es muy común ver que las aplicaciones utilizan bases de datos para almacenar toda la información (catálogos, usuarios, productos, facturas, etcétera). La interacción entre nuestro lenguaje de objetos y la base de datos, con sus tablas y columnas, puede ser tediosa y propensa a errores. En este sentido, una forma muy sencilla de mantener relacionados estos dos mundos es mediante un **ORM** (**Object Relational Mapping**, o Mapeo Relacional de Objetos). Debemos recordar que los frameworks que hacen **ORM** necesitan que se decoren las clases y sus atributos con anotaciones, para poder traducir los objetos a entradas en la base de datos y viceversa.

```
@Entity
@Table("EMPLEADO")
public class Empleado {

    @Id
    @Column("LEGAJO")
    private int numeroDeLegajo;

    @Column("NOMBRE")
    private String nombreYApellido;

    @ManyToOne
    @ForeignKey("LEGAJO_SUPERVISOR")
    private Empleado jefe;
    ...
}
```



HIBERNATE

Hibernate es un amplio y conocido proyecto de **ORM** en Java. Lo podemos encontrar en **www.hibernate.org**. Ofrece funcionalidad de punta a punta en el desarrollo de soluciones que se comunican con bases de datos, desde la creación de las tablas hasta el mapeo de los objetos de estas, soportando casi todas las bases existentes. Utiliza anotaciones para configurar el mapeo.

Aplicaciones web

Sabemos que desarrollar aplicaciones para la Web en Java puede tornarse tedioso si se realiza el trabajo desde cero, ya que se requieren muchos pasos y mucha configuración adicional. Muchos, entonces, se abocaron a la tarea de facilitar este procedimiento. Incluso existe una especificación Java, bastante reciente, que utiliza anotaciones y apunta a simplificar la creación de sitios y aplicaciónes web.

```
// si accedemos a /holamundo en nuestro sitio obtendremos el mensaje
    de bienvenida
public class HolaMundoResource {

      @GET
   @Produces("text/html")
   public String index() {
      return "<html><body><h1>Hola mundo!</body></h1></html>";
   }

}
```



RESUMEN

En este capítulo hemos aprendido qué son y para qué se utilizan las anotaciones. Cuál es su propósito y cuáles son sus limitaciones. Las anotaciones nos permiten agregar información al código de forma sencilla y práctica, ya que se encuentran juntos. Esta ventaja permite que la meta información y el código no estén sincronizados. Además, ofrecen la posibilidad de inspeccionarlas en tiempo de ejecución como en tiempo de compilación. Igualmente, antes de lanzarse a utilizar las anotaciones debemos preguntarnos si son la solución correcta a nuestro problema, o debemos mejorar nuestro diseño y bajar la información del nivel meta al nivel de los objetos de nuestro dominio.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Qué son las anotaciones?
2	¿Qué mecanismos similares se utilizaban antes?
3	Nombre algunas de las anotaciones que afectan al compilador.
4	¿Es posible que un método sobrescrito herede las anotaciones del método original?
5	¿Puede una anotación extender otra?
6	¿Cuáles son los tipos permitidos como propiedades de las anotaciones?
7	¿Se pueden definir métodos en las anotaciones?
8	Nombre algunos usos de las anotaciones.
9	¿Se heredan las anotaciones utilizadas en una interfaz en la clase que la implementa?
10	¿Cuándo conviene utilizar anotaciones?

ACTIVIDADES PRÁCTICAS

1	Crear un anotación para indicar que no se aceptan nulos, llamada @NoNulo.
2	Crear una clase de prueba como, por ejemplo, Persona y aplicar la anotación a sus atributos y a los argumentos de los métodos y constructores.
3	Crear una subclase de la anterior y sobrescribir cada método y constructor de tal forma que este realice las validaciones y luego llame al método o constructor original.
4	Agregar un método estático que permita instanciar objetos de la clase anterior y utilizarlo en lugar de hacer new.
5	Agregar un método que valide el estado del objeto (de sí mismo) basándose en las anotaciones.

*12

Técnicas y diseño

Hemos aprendido a lo largo de los capítulos anteriores sobre Java, cómo está formado y qué nos ofrece. Ahora es tiempo de aprender a utilizar esos conocimientos y esas herramientas de forma adecuada. Para realizar correctamente esta tarea debemos conocer ciertos conceptos de diseño muy útiles.



- **Inmutabilidad, Java Beans y la creación de objetos 226**
- **Inyección de dependencias e inversión de control 233**
- **Sobre la creación de objetos 237**
  - Método factoría 237
  - Factoría abstracta 239
  - Singleton 241
- **Evitar utilizar null 243**
- **Representar conceptos con objetos 245**
- **Resumen 247**
- **Actividades 248**

Inmutabilidad, Java Beans y la creación de objetos

Empecemos por tratar el tema de crear objetos que sean correctos y usables. Es muy común encontrar objetos que solamente tienen **getters** y **setters**. Esta clase de objetos es denominada **Java Beans** por razones históricas que ya nadie recuerda y tienen varios problemas de diseño. Primero, no tienen un comportamiento definido, solamente son contenedores de datos. En sí esto no es un problema, si no fuera por el hecho de que su comportamiento en realidad se encuentra en otros objetos que solamente tienen métodos para manipular estos **beans**. Por lo tanto estamos teniendo dos objetos, uno para contener la información y otro para operar sobre esta, básicamente no estamos utilizando el paradigma de objetos y volvimos a programar en C. Recordemos que los objetos encapsulan su estado (información) y operaciones (métodos) en un solo lugar. Dividir esto en dos es crearse problemas innecesariamente. Si encontramos clases que solo definen atributos con setters y getters, debemos buscar las clases que operan sobre la primera y tratar de mover la funcionalidad de una clase a otra, así mantenemos juntos el estado y la funcionalidad.

> ES COMÚN ENCONTRAR OBJETOS QUE SOLO TIENEN GETTERS Y SETTERS

Debemos recordar que es una falencia común en el ámbito laboral encontrarse con desarrolladores que programan de esta forma, que puede considerarse como completamente errónea. Por esta razón es necesario tener presente que no debemos caer en esta práctica.



PROGRAMANDO CON ESTRUCTURAS

Debemos tener en cuenta que el estilo de programación con estructuras que utiliza objetos que solo contienen datos, y luego, en otro objeto o clase, tiene métodos para manipularlos y utilizar estos objetos viene de **C** y de otros lenguajes **procedurales**. En estos lenguajes no había objetos y la forma de trabajo era esta, donde primaban los procedimientos que usaban datos.

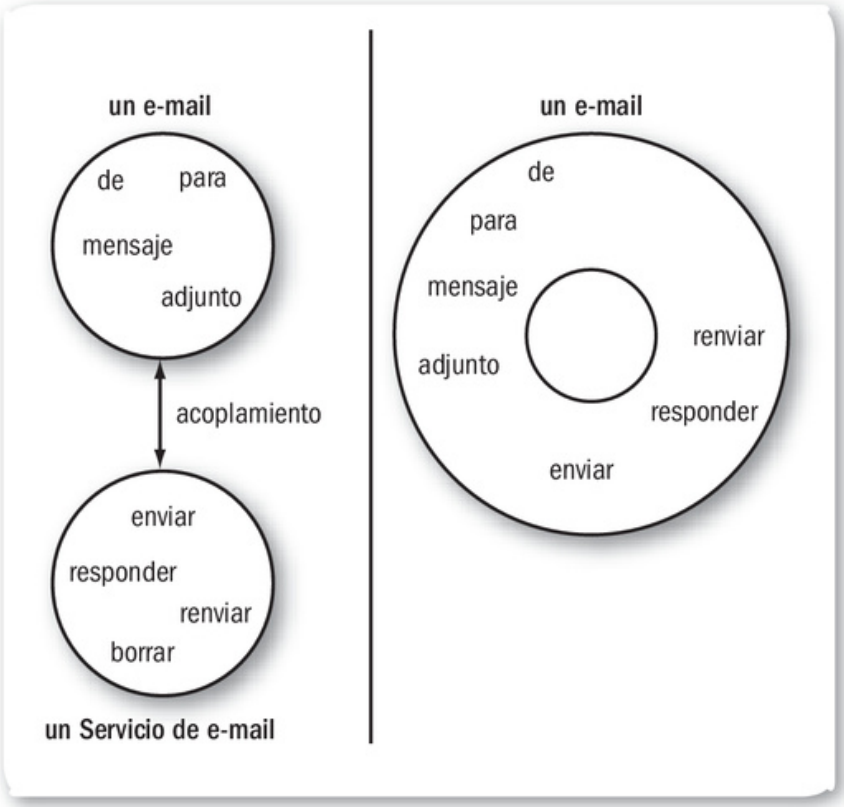


Figura 1. Acoplamiento entre el objeto java bean y el objeto que tiene la lógica para manipularlo.

Por ejemplo, esta sería una hipotética clase (pero no tanto, ya que **Spring** tiene un diseño similar para esto) que modela un e-mail. Es una clase "tonta", no hace nada, salvo contener datos.

```
public class Mail {

    private String to;
    private String from;
    private String body;

    public String getTo() {
        return to;
    }
    public void setTo(final String to) {
```

```
        this.to = to;
    }
    public String getFrom() {
        return from;
    }
    public void setFrom(final String from) {
        this.from = from;
    }
    public String getBody() {
        return body;
    }
    public void setBody(final String body) {
        this.body = body;
    }
}
```

Y esta es la clase que nos permite manipular y operar un e-mail. Vean cómo, sin esta clase, la anterior es inútil.

```
public class MailService {

    public void enviarMail(final Mail mail) {
        ...
    }
    public void reenviarMail(final Mail mail, final String destinatario) {
        ...
    }
```



BENEFICIOS DE LA INMUTABILIDAD

Los objetos inmutables tienen varios beneficios: son fáciles de crear, probar y usar; son **thread-safe**, no requieren sincronización; no requieren ser copiados defensivamente cuando son atributos y se los devuelve en un método; son excelentes candidatos para estar en un **Set** o como claves de un **Map**; su invariante de clase se valida una sola vez al construirlos y nunca están en un estado inválido.

```
    public void responderMail(final Mail mail, final String respuesta) {
        ...
    }

        ...
}
```

¿Tiene sentido mantener dos cosas cuando podríamos mantener solo una? Si sabemos que una clase es inútil, ¿para qué la tenemos? Estas son algunas de las preguntas que debemos hacernos cuando nos enfrentamos a este tipo de diseño.

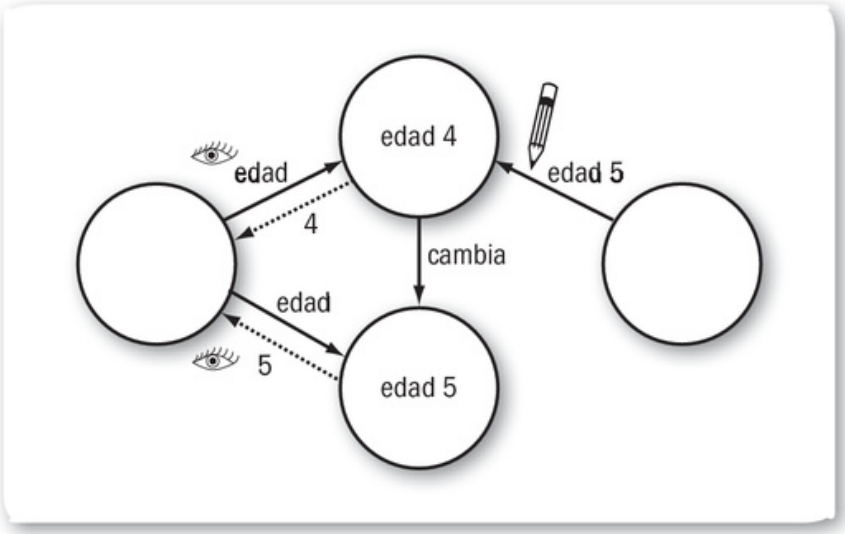


Figura 2. En este diagrama podemos apreciar cómo pueden producir errores los objetos mutables.

Otra característica pobre de los objetos puramente de datos es que, casi en su mayoría, son mutables, o sea que pueden cambiar en cualquier momento. Si bien esto puede ser lo que queramos, en general, la mayoría de los objetos del mundo real son inmutables. Supongamos que tenemos el objeto fecha **10 de septiembre de 2010**; si pudiéramos cambiar la propiedad **día** de tal objeto por 20, este dejaría de modelar la fecha inicial y el resto del código seguiría creyendo que es el 10. Este tipo de confusiones producen errores

que son extremadamente difíciles de encontrar y arreglar. Sería más fácil si desde el principio el objeto **fecha** no pudiera ser modificado. Si queremos cambiar de fecha, simplemente creamos otra fecha, sin afectar al resto del código que depende de la fecha original. Esto es muy importante, especialmente en ambientes de muchos procesos en paralelo, como lo es una aplicación web, que atiende varios clientes simultáneamente. También al hacer los objetos inmutables nos protegemos de que al devolver un colaborador interno como respuesta de un método, no estemos expuestos a que accidentalmente un cliente modifique tal objeto y que esto nos afecte. Como regla general, deberíamos hacer que todos los objetos que creamos sean inmutables, a menos que sea realmente necesario que así no sea. Para lograr objetos inmutables en Java debemos utilizar el modificador final en los atributos e inicializarlos utilizando el constructor (o directamente en la definición de los atributos).

> COMO REGLA GENERAL, TODOS LOS OBJETOS QUE CREAMOS DEBERÍAN SER INMUTABLES

```
public class Punto {
    private final double x;
    private final double y;

    public Point(final double x, final double y) {
        this.x = x;
        this.y = y;
    }

    public double x() {
        return x;
    }

    public double y() {
        return y;
    }

}
```

Es necesario tener en cuenta que en objetos que requieran una mayor inicialización, podemos utilizar setters privados y tratar de usarlos solamente desde un único punto, el constructor.

No solo debemos tratar de controlar los cambios que pueden afectar a un objeto determinado, sino que también tenemos que cuidarnos de que solamente se creen objetos que sean válidos. Reflexionemos, ¿nos interesa tener una fecha inválida? ¿De qué nos sirve? ¿Hay fechas inválidas en la realidad? Por definición una fecha es válida siempre, si no, no existe. En este sentido, debemos saber que esta restricción tiene que ser modelada también y por lo tanto, debemos asegurarnos de crear objetos correctos desde el inicio de su vida.

Supongamos que dejamos que se creen fechas sin inicializar y que podemos ir cambiando las propiedades de día, mes y año.

```
Fecha fecha = new Fecha();
fecha.setDia(29);
fecha.setMes(2);
fecha.setAño(2011);
// ERROR fecha no válida!
// arreglémosla
fecha.setAño(2012);
// ahora sí existe
// también podríamos haber hecho
Fecha.setDia(28);
```

Recordemos que la validación se tiene que realizar en cada paso, ya que ante cualquier acción puedo tener una fecha inválida. Además hasta que no está inicializada, la validación no tiene sentido. Esto es mucha lógica que tiene que estar respaldada por mucho código, solamente para tener fechas válidas, algo que podríamos conseguir tan solo inicializando y validando en el constructor.

```
// ERROR fecha inexistente
Fecha fecha = new Fecha(29, 2, 2011);
// fecha correcta
Fecha fecha = new Fecha(29, 2, 2012);
```

En el constructor hacemos las validaciones pertinentes.

```
public class Fecha {
    ...
    public Fecha(final int dia, final int mes, final int año) {
        // inicializo las variables y verifico
    if(!valida())
        throw new FechaInvalidaException();
    }
    ...
}
```

Incluso sería mejor modelar correctamente las entidades de mes y año, y las relaciones entre ellos, tal como lo hicimos en un ejercicio de un capítulo anterior. Volvamos un segundo al ejemplo del punto donde definimos un constructor que acepta el componente **x** y el componente **y** (las coordenadas de este). Este constructor es confuso en el sentido que tenemos que asociar los nombres de los argumentos para entender qué hace. ¿Va **x** primero y después **y**, o primero **y**, y después **x**? ¿Son coordenadas cartesianas o polares? La semántica del constructor debería estar dada por este. Lamentablemente en Java los constructores se llaman igual que la clase y por lo tanto lo único que distingue a unos de otros son los parámetros. En consecuencia a veces es recomendable utilizar métodos estáticos para la construcción.

```
public class Punto {
    public static Punto conXY(final double x, final double y) {
        // creamos un punto con tales coordenadas
    }

    public static Punto conDistanciaYAngulo(final double distancia, final double
        angulo) {
    // creamos un punto basado en las coordenadas polares
    }
    ...
}
```

Los métodos constructores ya nos dan la semántica de los argumentos y no importan los nombres de estos. Una ventaja adicional de este tipo de aproximación es que no nos interesa cómo se representa internamente el punto, solo cómo lo creamos.

Lo importante de esta sección es que debemos restringir las modificaciones a los objetos lo más que podamos y de acuerdo a cómo se toman esas modificaciones en el dominio del problema. Evitar, en lo posible, los **setters** y las clases **tontas** que solo contienen datos. Finalmente tratar de darles semántica a los constructores o, si tenemos varias formas de creación para un mismo tipo de objeto, utilizar métodos estáticos.

Inyección de dependencias e inversión de control

En el capítulo anterior mencionamos algunos frameworks de inyección de dependencias. Veamos, ahora, con mayor detalle, de qué se trata este concepto y cómo se relaciona con otro, la inversión de control. La inversión de control trata de distribuir las responsabilidades de un sistema en varios objetos, de forma que cada uno tenga una responsabilidad clara, bien definida y con un único propósito. Este concepto está asociado generalmente a la inicialización de colaboradores (de ahí que se lo relacione solamente a la inyección de dependencias), pero aplica a todo tipo de responsabilidad. Por ejemplo, si tenemos un objeto con varios atributos y varios métodos, y solo una parte de esos métodos es lo único que actúa sobre una parte de los atributos. Claramente este objeto está teniendo dos



VÁLIDO POR NATURALEZA

Supongamos que utilizamos un **int** para modelar algo que siempre debe ser positivo, debemos usarlo en todos lados donde un parámetro sea positivo, una y otra vez. En vez de eso, podemos crear un nuevo tipo que naturalmente represente a los números positivos. Así no ensuciaremos el código con validaciones repetitivas y además abstraeremos el nivel de nuestro programa.

responsabilidades y podríamos dividirlo fácilmente en dos objetos, donde cada uno al ser responsable de su funcionalidad colabora para proveer la funcionalidad original.

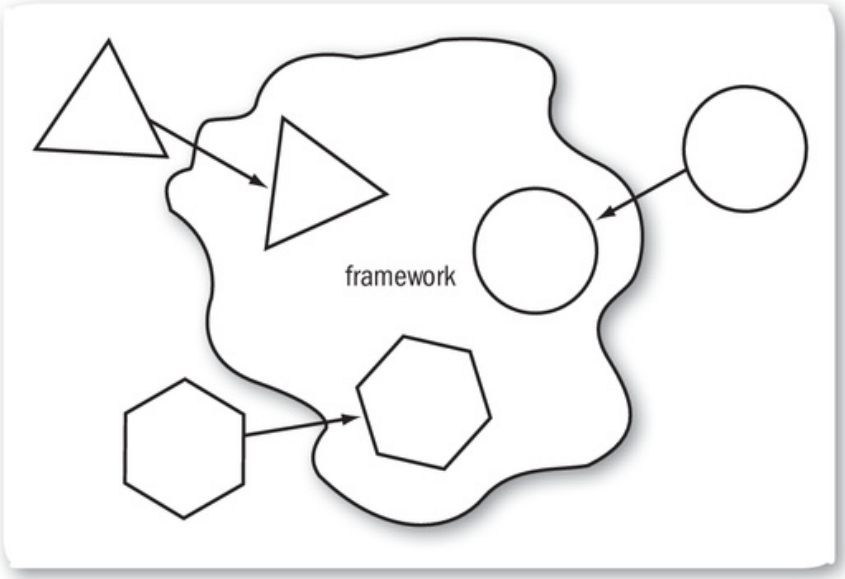


Figura 3. Este diagrama nos muestra la relación entre un framework y las partes de nuestro código.

La inversión de control es la diferencia clave que existe entre una librería y un framework. Cuando usamos una librería, nosotros nos encargamos de manejar el ciclo de vida de sus objetos y de activarlos (utilizarlos). En cambio, en un framework, el que está en control de la ejecución es el framework y él se encarga de activar nuestros objetos cuando lo cree necesario. Todos los frameworks tratan sobre la inversión de control aunque, popularmente, solo se utiliza este



DOMAIN SPECIFIC LANGUAGES

Esta novedad apunta a crear protocolos que permitan ser encadenados y que puedan formar frases que se parezcan lo más posible a enunciados del lenguaje humano. De esta forma es más natural pasar del dominio al modelo. Un ejemplo sería **Punto.conX(x).yConY(y)** para construir un punto. Esto es claro, pero se requieren protocolos más complicados para armar las frases.

concepto para tratar los frameworks de inyección de dependencia o de configuración de objetos. Como ya dijimos la inyección de dependencias trata sobre cómo configurar los colaboradores de un objeto. Esto lo debemos hacer aunque no utilicemos un framework específicamente para esto. Lo que necesitamos es uno o más objetos que se encarguen de crear las dependencias, crear el objeto que nos interesa y enlazarlos. Los frameworks de inyección de dependencias no solamente manejan la creación y configuración de los objetos y sus dependencias, sino que agregan más funcionalidades que permiten establecer cuándo y cómo se deben crear o, si es posible, reusar instancias para no crear objetos costosos de más.

> CONSIDEREMOS QUE NO TODOS LOS COLABORADORES DEBEN SER EXTERNALIZADOS

Hay que considerar que no todos los colaboradores deben ser externalizados. Solamente deberíamos tener como dependencias a aquellos colaboradores que existan como concepto fuera del objeto y que no están atados al ciclo de vida de este último. Por ejemplo el mes de febrero existe como objeto más allá de si es colaborador para la fecha 15 de febrero de 2009. En cambio, una colección para guardar los elementos que componen una figura debería permanecer oculta dentro del objeto figura, ya que es un detalle de implementación que no le interesa a un cliente.

A continuación podremos ver cómo serían ambos ejemplos en código. Primero, en el caso donde el colaborador es externo y no depende de nuestro objeto para existir.

El objeto que se encarga de representar al mes existe mas allá de que sea o no utilizado para un fecha en particular. En un segundo caso el colaborador solo existe mientras existe su contenedor.



MARTIN FOWLER

Martin Fowler es un famoso orador sobre patrones de diseños y arquitectura. En su sitio, encontraremos muchos artículos interesantes. Entre ellos un análisis de los frameworks de inyección de dependencias e inversión de control. Lo podemos ver en **http://martinfowler.com/articles/injection.html**, el sitio tiene varios ejemplos sobre la filosofía de los principales frameworks.

```
public class Fecha {
    ...
    private final Mes mes;
    ...
    public Fecha(final Dia dia, final Mes mes, final Año año) {
        ...
    }
    ...
}
```

En el segundo caso, el colaborador es interno y solamente existe mientras está vivo el objeto que lo contiene.

La existencia del colaborador interno no tiene sentido mas allá del tiempo de vida del objeto que lo contiene.

```
public class Figura {
    private List<Forma> formas = new ArrayList<Forma>();
    ...
    public Figura() {
        ...
    }
    ...
    public void agregarForma(final Forma forma) {
       formas().add(forma);
    }
    ...
}
```



ELEGIR BIEN QUÉ FRAMEWORK USAR

Es muy importante tener en cuenta que si queremos utilizar un framework de inyección de dependencias debemos investigar y ver qué requiere cada uno de nuestros objetos para poder crearlos y configurarlos. Si nos interesa elegir cuál de estos frameworks usar y, seguramente encontraremos uno que se acomoda a nuestro estilo y a nuestras necesidades, ya que las alternativas disponibles son variadas.

Sobre la creación de objetos

Ya vimos que debemos delegar la responsabilidad de creación y configuración de los objetos en otros, ahora avancemos un poco más en las distintas estrategias para esta tarea. Existen varios patrones aceptados por la comunidad de programadores sobre cómo enfrentar el problema de la creación. Los patrones de diseño son soluciones ampliamente aceptadas para problemas recurrentes en los sistemas. Para que surja un patrón, este debe estar validado por varios sistemas que hayan solucionado un problema por medio de este. Esta solución luego se refina en un patrón. Existe un amplio catálogo de patrones conocidos, que atacan distintos tipos de problemas. Este es un tema que merece una atención extra de nuestra parte.

Método factoría

El primero de estos patrones ya lo hemos visto y usado varias veces en los ejercicios, y es el que está basado en utilizar un método que cree el objeto que corresponde. El método puede ser tanto de instancia como estático y permite que el cliente no sepa efectivamente qué implementación se está utilizando. El método es el encargado de decidir cuál de las implementaciones corresponde. Si el método es de instancia, es posible construir jerarquías que sobrescriban el comportamiento de este según convenga.

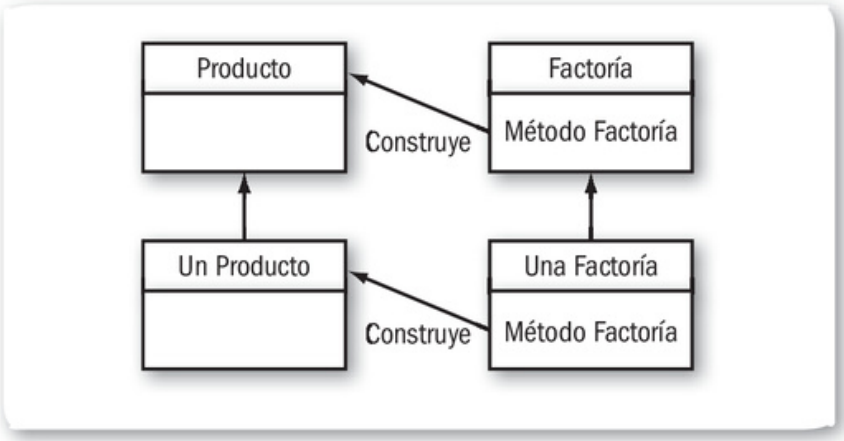


Figura 4. Diagrama teórico de clases.

La clase **Collections** de Java es un buen ejemplo de este tipo de patrón, donde los distintos métodos crean objetos de los cuales no sabemos a ciencia cierta qué implementaciones tienen.

```
public class Collections {
    ...
    public static <E> List<E> emptyList() {
        ...
    }
    ...
    public static <E> List<E> singletonList(E o) {
        ...
    }
}
```



Figura 5. Christopher Alexander, famoso arquitecto, es el responsable de originar la tendencia de los patrones.

PATRONES DE DISEÑOS

En el sitio **www.oodesign.com** encontraremos un amplio catálogo de patrones para lenguajes orientados a objetos. En él se encuentran los famosos patrones del **GoF** y algunos otros muy comunes. Podremos acceder a toda la información sobre cada patrón, los esquemas, la motivación, cuándo hay que aplicarlos, los pros y los contras, y concejos para la implementación.

Factoría abstracta

Ahora supongamos que tenemos una familia de objetos para crear y hay varias implementaciones de cada una. Por ejemplo, imaginemos que tenemos muebles para una sala: sillas, mesas, sillones y lámparas. Ahora tenemos distintos estilos para estos muebles: clásico, retro y moderno. Entonces tenemos la jerarquía de muebles y estilos, y ahora queremos poder obtener los muebles de un mismo estilo de una forma que no tengamos que especificarlo siempre. Para ello generamos otra jerarquía de objetos que nos devuelva las sillas, mesas, sillones y lámparas que corresponden a ella.

Entonces cada jerarquía corresponde a un estilo distinto, con muebles pertenecientes a ese estilo en particular.

```
...
public void decorarCon(Estilo estilo) {
    setMesa(estilo.mesa());
    setSillas(
        estilo.silla(),
        estilo.silla(),
        estilo.silla(),
        estilo.silla()
    );
    setUnSillon(estilo.sillon());
    setOtroSillon(estilo.sillon());
    ...

}
...
```



SOBRE LOS PATRONES DE DISEÑO

Los patrones de diseño están basados en los estudios del arquitecto **Christopher Alexander** a finales de la década del 70. Alexander propuso una forma de reutilizar conceptos para construir a cualquier escala, y para esto se basó en la observación de muchas ciudades y edificios, de los que abstrajo las formas básicas y comunes. Luego, la computación tomó estas ideas y las aplicó a su propia disciplina.

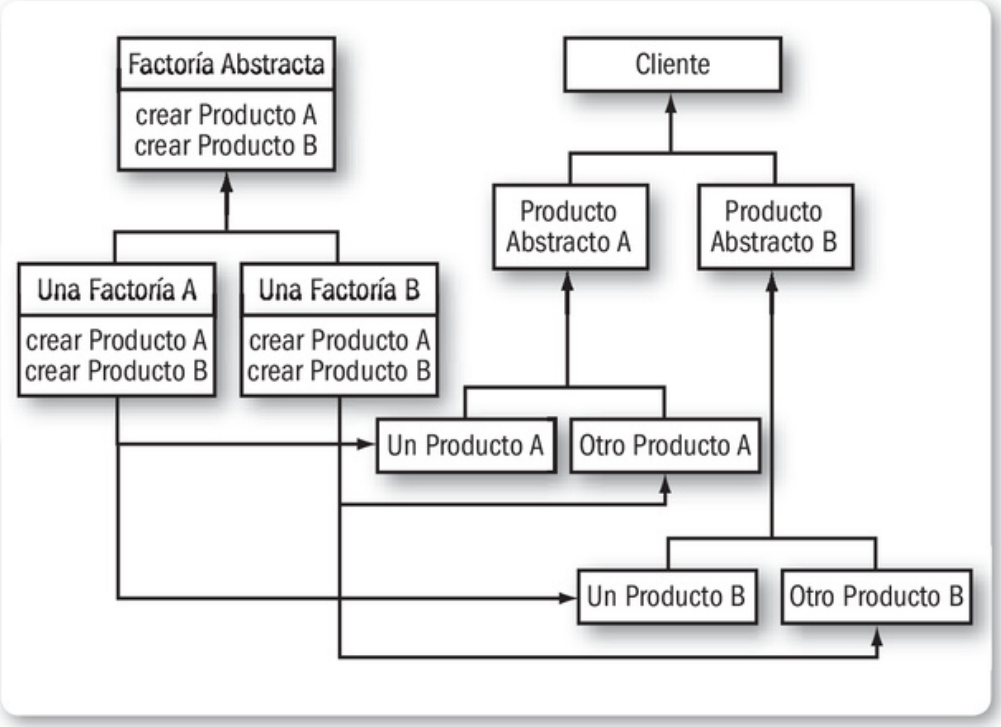


Figura 6. En esta imagen podemos ver un diagrama teórico de clases de varias factorías con varios productos.

Debemos tener en cuenta que de esta forma cuando queremos todo los muebles de un mismo estilo, solamente tenemos que utilizar el objeto constructor que corresponde.

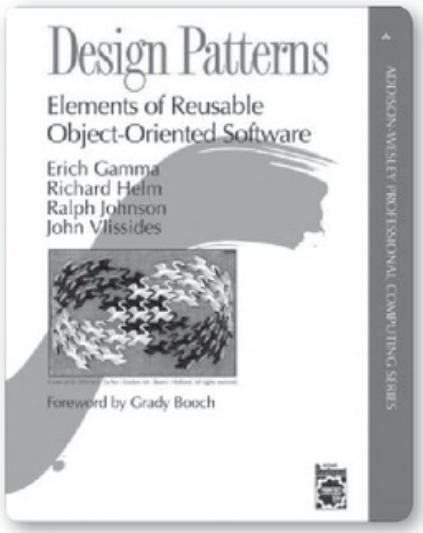


Figura 7. Aquí vemos la cubierta del famoso libro de patrones del Gang of Four.

Singleton

Uno de los patrones más comunes de creación en Java es el **singleton**, que significa que solamente existe una instancia de una determinada clase. Este tipo de patrón es usado cuando tenemos un recurso costoso y queremos que sea reutilizado en lugar de crear nuevas instancias. Lamentablemente es muy común encontrar este patrón mal implementado ya que se utiliza un método estático para devolver la instancia que hace que el objeto sea global (con los problemas que traen los objetos globales).

```
public class BaseDeDatos {
    // se pone el constructor privado
    private BaseDeDatos() {
    }

    // tenemos la instancia estática
    private static BaseDeDatos instancia;

    // creamos una método para obtener la instancia única
    public static BaseDeDatos getInstancia() {
        ...
    }
    ...
}
```

Primero, hemos puesto un objeto en el alcance global, al hacer que potencialmente cualquier código pueda tener acceso a él. Segundo, hemos esparcido el conocimiento de que estamos tratando con una



GANG OF FOUR O EL GRUPO DE LOS CUATRO

Así se conoce a los cuatro autores del libro más conocido sobre patrones de diseño y el que marcó el inicio de una tendencia. Ellos son **Erich Gamma**, **Richard Helm**, **Ralph Johnson** y **John Vlissides**. Su libro, **Design Patterns: Elements of Reusable Object-Oriented Software**, es de cabecera y debe ser leído y consultado constantemente para mantener presentes sus ideas.

única instancia por todos lados (donde se usa). Tercero, supongamos que queremos cambiar esto porque nos dimos cuenta de que en realidad no es necesario que sea una única instancia y queremos eliminar el **singleton**. Tenemos que realizar la modificación en todos los lados donde lo usemos, lo cual nos generará un enorme trabajo. Que sea una única instancia es implementativo y deberíamos evitar que los clientes se enteren de tal decisión.

CONVIENE UTILIZAR UN FRAMEWORK PARA MANEJAR LOS SINGLETONS

Podemos utilizar otro nombre para el método que devuelve la instancia. Luego, en el código cliente deberíamos tratar de utilizar lo menos posible la referencia directa a la clase y al método, y solo usarla al principio para configurar los objetos que lo requieran, para que luego se pase como parámetro, tanto de construcción como para algún método. Tercero, podemos utilizar alguno de los frameworks de inyección de dependencias para que administre una única instancia y así obtener el mismo fenómeno del **singleton** pero sin codificarlo. Una última forma de atacar este problema es con una clase interna que sea la que va a ser efectivamente el **singleton**, y que aquella clase, tenga el **singleton** y lo use.

```
public class BaseDeDatos {

    private static Singleton singleton;

    public BaseDeDatos() {}

    public void guardar(Dato dato) {
        singleton.guardar(dato);
    }

    ...

    private class Singleton extends BaseDeDatos {
        ...
    }
}
```

Con todo lo visto hasta aquí podemos darnos cuenta de que tenemos varias instancias que solamente delegan todo comportamiento en el **singleton**. Así, estas instancias son útiles mientras solamente existe una única instancia que se presenta como costosa.

Evitar utilizar null

null es un problema. No es un objeto, es un elemento extraño. Si tratamos de utilizar una referencia que está apuntando a **null** obtendremos un error. Lo mejor es reducir su uso. ¿Qué significa esto? Por ejemplo, si tenemos una colección vacía, no usemos **null** para esto, sino "una colección vacía", incluso en métodos de búsqueda. De esta manera el cliente no tiene que preguntar si la respuesta es **null** o no. También es importante no tener atributos nulos, ni tener parámetros de más que tengan que ser pasados como **null** cuando son opcionales. Muchos métodos que reciben varios parámetros tienden a permitir que algunos sean nulos, esto complica el código. Es mejor agregar métodos con la cantidad de parámetros obligatorios correcta. Si diseñamos correctamente, podemos disminuir el uso de **null**, y el consecuente código para preguntar si algo es **null** o no.

```
File directorioPadre = directorio.getParentFile();
// tengo que preguntar siempre
if(directorioPadre != null) {
    // hacemos algo con el directorio padre
}
```

LOS PATRONES DE DISEÑO

Los patrones de diseño no solo son un compendio de soluciones probadas para un determinado tipo de problema, sino que, lo más importante es que definen un lenguaje de comunicación más rico. Gracias a ellos es posible comunicar un conjunto de ideas respecto de un diseño utilizando solamente el nombre de un patrón. Ellos enriquecieron el lenguaje que utilizamos los programadores.

Todo código cliente de este tipo de método tiene que preguntar indefectiblemente si es **null** o no. Sería mucho más fácil si movemos ese código a la clase que corresponde y que los clientes solamente provean el comportamiento específico, o utilizar un iterador y usar el **foreach**.

```
// código hipotético
for(File padre : directorio.getParentFile()) {
    // hacemos algo con el directorio padre
}
```

Utilizar el **foreach** para este tipo de cosas es práctico, ya que no se ejecuta si no hay nada. También, podemos pasar una clase anónima a un método que aplique dicha clase al objeto que puede ser nulo y así verificar si es nulo o no está en un solo lugar.

```
directorio.getParentFile(new With<File>() {
 @Override public void do(File parent) {
    // hacemos algo con el directorio padre
 }
});
```

Otra opción que existe es la posibilidad de modelar el vacío, el no objeto, explícitamente. Esto se conoce como el patrón **Null Object**. De esta forma creamos un objeto polimórfico que representa la nada, con el tipo de objetos que necesitamos que sea así.

```
public class CuentaNula extends Cuenta {

    @Override
   BigDecimal saldo() {
        return BigDecimal.ZERO;
    }

    @Override
```

```
    void transferirA(Cuenta destino, BigDecimal monto)
{
        return;
    }

    @Override
    Cuenta combinarCon(Cuenta otraCuenta) {
        return otraCuenta;
    }

    ...
}
```

Estos objetos nulos deben devolver objetos que representen nulos en todos los métodos que requieran respuesta y, en consecuencia, no hacer nada en todos ellos. De esta forma la semántica de nulo se va extendiendo por el sistema en vez de arrojar error. Los errores se deberían atrapar con test unitarios bien hechos.



Representar conceptos con objetos

Y una vez más volvemos a esta idea, a este concepto, que deberíamos seguir siempre que estamos diseñando y programando. Como bien venimos aprendiendo desde el primer capítulo, al modelar una determinada situación, tenemos que tratar de crear una relación, uno a uno, entre los conceptos de la situación por modelar y el modelo. Esta forma de modelar nos permitirá fácilmente verificar que las reglas que gobiernan el problema se cumplan de igual manera en el modelo. Durante el libro hemos visto ejemplos de los frutos que nos da esta técnica. En el ejercicio de El Juego de la Vida, vimos como **reificando** los tipos de células y los distintos tipos de vecindarios obtuvimos una implementación sencilla, clara y extensible, sin perder velocidad. Cuando diseñamos un reemplazo para la clase **Date**, modelamos los

meses y los años como entidades propias en vez de utilizar números. De esta forma obtuvimos un conjunto de clases flexibles y que forzaban un buen uso de ellas y minimizaban errores.

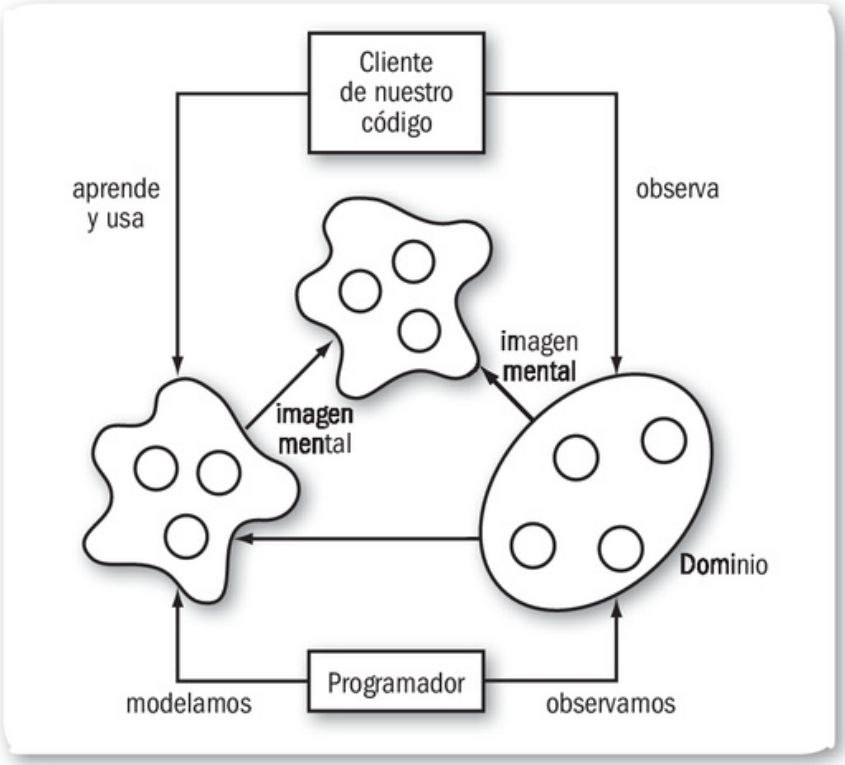


Figura 10. En esta imagen podemos ver el dominio y las distintas imágenes que hay de este, siendo una de ellas nuestro modelo y la otra la del cliente de nuestro modelo.

La idea básica es simple, si algo existe en el dominio del problema, entonces tiene que existir en el modelo de nuestra solución (es un buen momento para releer el primer capítulo). No debemos tener miedo de crear nuevas clases que representen conceptos nuevos, si algo es distinto, debe ser modelado de forma distinta. Obviamente cuanto ahondemos en esta tarea dependerá del nivel de complejidad del domino.

Esta técnica está íntimamente relacionada con los **test unitarios**. Los test nos permiten experimentar con la interfaz, el protocolo de nuestros objetos desde el principio. Vemos cómo se relacionan y colaboran para proveer una funcionalidad. Con esto podemos ir ajustando la **API** para que sea amigable al usuario, que comunique

apropiadamente las ideas y sea sencilla de entender y, por lo tanto, de utilizar. Tenemos que tener en mente que cuando trabajamos con un dominio en particular, con sus reglas y propiedades, en nuestra cabeza tenemos una idea determinada y aproximada de lo que en realidad es. Ahora, si sumamos a un cliente, este tiene otra idea, que si bien será similar a la nuestra, no será igual y tampoco será igual al dominio real. Por lo tanto tenemos que esforzarnos en acortar la brecha de las diferencias de entendimiento entre nuestro modelo, lo que piensa el cliente y lo que en realidad es el dominio.

Por ese motivo hemos visto a lo largo de los capítulos técnicas y herramientas para poder acercarnos a dominio y así unificar conceptos con el cliente. En teoría el cliente debe conocer en profundidad el dominio del problema. En principio vimos que el paradigma nos ofrecía esta visión de objetos que interactúan, muy propicia para la comunicación con otra persona. Después tenemos a los test unitarios y el desarrollo utilizándolos (TDD, Test Driven Development), que nos permiten desde el principio del desarrollo, atacar la interfaz de nuestro sistema de forma que sea lo más comunicativa y simple. Finalmente, todos los conceptos e ideas de este capítulo aumentan nuestra caja de herramientas para que a la hora de diseñar lo hagamos bien.



RESUMEN

En este capítulo hemos conocido y analizado varias técnicas y herramientas que nos ayudaran a diseñar mejor y así obtener mejores sistemas. Esto nos entregará como resultado que el sistema será comprensible, comunicará claramente lo que hace y, por lo tanto, será más flexible (adaptable a modificaciones) y sostenible. Como parte de estas técnicas hemos visto algunos patrones de diseño funcionales. Como vimos, se trata de un tema muy interesante y que nos ayudara en nuestros desarrollos.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Qué son los Java Beans?
2	¿Por qué no son una buena manera de desarrollar?
3	¿De qué forma aseguraría la inmutabilidad de sus objetos?
4	¿Cuál es la razón de que sea tan importante este concepto?
5	Defina inversión de control.
6	¿Qué nos ofrece un framework de inyección de dependencias?
7	¿Qué son los patrones de diseño?
8	¿En qué influyen los test unitarios durante la etapa de diseño de una API?
9	¿Qué consideraría como un buen diseño?
10	¿Por qué debemos tener un lenguaje común con el cliente?

ACTIVIDADES PRÁCTICAS

1	Modele una fecha como Java Bean y resuelva los días del mes de febrero. ¿Qué diferencias hay con el modelo de fecha que hicimos anteriormente?
2	Modele una lista inmutable.
3	Modifique el ejercicio de capítulos anteriores referido al uso de colecciones inmutables.
4	Modele un subsistema de archivos y directorios sin utilizar null, creando todos los conceptos necesarios. Utilizar en el fondo, el API de Java.
5	Busque y lea más sobre los patrones de diseño.

*Ap1 –

Reflexión

Java nos ofrece herramientas para que en tiempo de ejecución podamos estudiar la estructura de nuestro código, de nuestros objetos vivos, y de esta forma podamos modificar comportamientos e incorporar nuevas clases, mediante el envío de mensajes a los metaobjetos que modelan los componentes de un programa.

- ¿Qué es la reflexión? 250
- Las clases 251
- Métodos 255
- Constructores 261
- Atributos 262
- Modificadores 264
- Interfaces 265
- Clases anidadas 265
- Arrays 268
- Los ClassLoaders 277
- Resumen 277
- Actividades 278



¿Qué es la reflexión?

Java es un lenguaje reflexivo. Eso significa que los programas Java pueden reflejar su propia ejecución y estructura. Los **metaobjetos** del sistema se pueden **reificar** como objetos ordinarios, que pueden ser consultados e inspeccionados como cualquier otro objeto. Los metaobjetos de Java son: clases, métodos, atributos, constructores, modificadores y paquetes. Este procedimiento de reflexión también es conocido como **introspección**. Java ofrece soporte limitado para la modificación de los objetos mediante reflexión. Permite modificar atributos, estáticos y de instancia, esto se conoce como **intercesión**. A un programa que manipula otro se lo llama **metaprograma**.

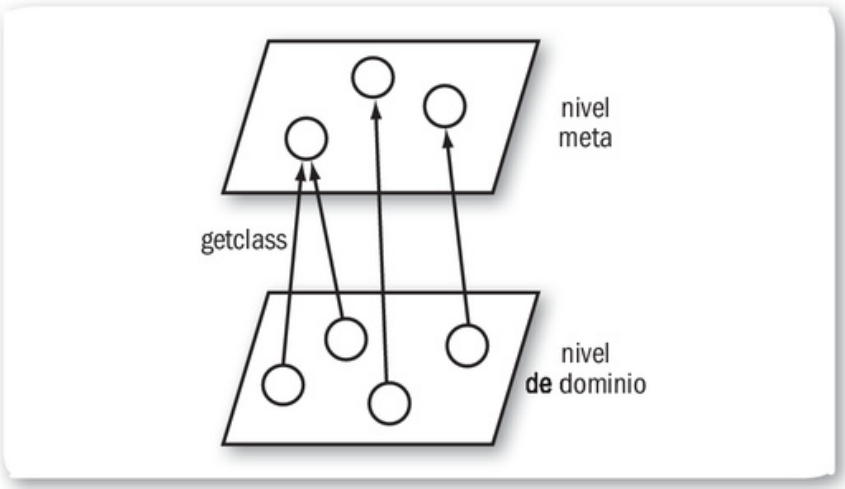


Figura 1. Aquí vemos la transición entre el mundo de los objetos del dominio y el de los metaobjetos.

BEHAVIORAL REFLECTION

Es interesante tener en cuenta que con este nombre se conoce a un tipo de reflexión que permite realizar la consulta y posterior modificación del comportamiento del sistema. Dicho de otra forma, permite inspeccionar qué hacen los métodos y modificarlos en tiempo de ejecución. Lamentablemente, Java solamente soporta la primera funcionalidad, con la que podemos obtener los métodos y mirarlos, pero no podemos modificar su **bytecode** para cambiar su funcionamiento.

Las clases

En Java todo objeto (no los primitivos) pertenece a una clase. Estos objetos son los que se llaman tipos referenciados, ya que siempre se tiene una referencia al objeto. Los tipos primitivos se tratan por separado, y no son objetos ni se los puede referenciar. Volviendo a las clases, veamos cómo obtener el objeto que representa a una clase. Hay varias formas, pero la dos más sencillas son pidiéndole a un objeto su clase mediante el método **getClass()** o utilizando el literal de la clase. Ahora vemos cómo acceder mediante la primera opción:

```
// obtengo la clase de un objeto
String texto = "hola mundo!";
Class<? extends String> string = texto.getClass();

assertSame("hola mundo!".getClass(), String.class);
```

Debemos sabe que el método **getClass()** está definido en la clase **Object** y por lo tanto todos los objetos del sistema lo tienen. Ahora veamos un ejemplo utilizando la segunda opción.

```
// uso el literal de una clase
Class<? extends Date> date = Date.class;
```

Las clases son tipos referenciados y son, a su vez, objetos. Por lo tanto, si son objetos, deben ser instancias de alguna clase. En efecto las clases son instancias de otra clase llamada **Class**.

```
assertSame(String.class.getClass(), Class.class);
```

Ahora si existe una clase llamada **Class** y, como todas las clases, es un objeto, entonces debe ser instancia de alguna clase, que a su vez sería un objeto y sería una instancia de otra clase; y así hasta el infinito. Para resolver esta recursividad infinita, se hace que el objeto **Class** sea instancia de la clase **Class**, o sea **Class** es instancia de sí misma.

```
assertSame(Class.class.getClass(), Class.class);
```

Ahora recordemos que toda clase, en definitiva, hereda de **Object**, ya que todos los objetos son **Object**. La forma de ir subiendo en la jerarquía de clases es mediante el mensaje **getSuperclass()**, que entienden todas las clases (está definido en **Class**).

```
assertSame(String.class.getSuperclass(), Object.class);
```

La superclase de **String** es **Object** ya que hereda directamente. En otros casos, como con la clase **ArrayList**, si utilizamos el método getSuperclass() hasta alcanzar Object veríamos las clases **AbstractList**, luego **AbstractCollection** y finalmente **Object**.

Debemos tener en cuenta que esta cadena, que se encuentra formada por las relaciones de herencia entre las clases, es fácilmente navegable con los métodos que nos hemos encargado de presentar. A continuación podemos ver un ejemplo de lo que acabamos de comentar:

```
// ArrayList → AbstractList
assertSame(ArrayList.class.getSuperclass(), AbstractList.class);

// AbstractList → AbstractCollection
assertSame(AbstractList.class.getSuperclass(), AbstractCollection.class);

// AbstractCollection → Object
assertSame(AbstractCollection.class.getSuperclass(), Object.class);
```



HERENCIA MÚLTIPLE

Viendo lo sencillo que es el metamodelo (el modelo de las clases) de Java, ahora piensen cómo sería la navegación de las superclases en presencia de la herencia múltiple. Sería todo un problema, ya que podría aparecer la misma clase varias veces. Además tendríamos que planear la forma de navegarlas. Este es otro punto a favor de la herencia simple por sobre la múltiple.

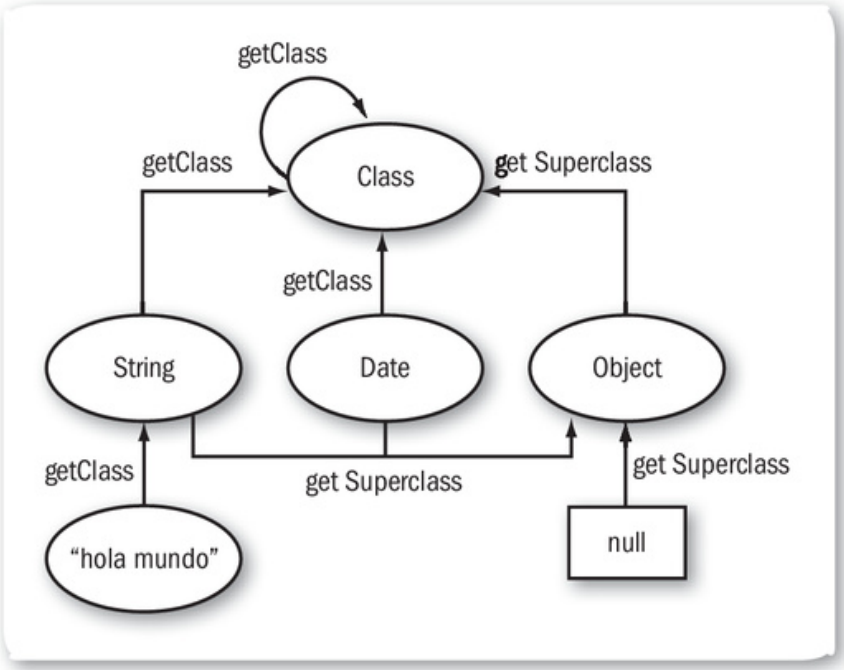


Figura 2. Relaciones basadas en **getSuperclass()** y **getClass()** entre algunas clases y las clases **Class** y **Object**.

¿Y la superclase de **Class**? La superclase de **Class** es **Object** ya que hereda directamente. Ahora entonces, **Object** es una clase, por lo tanto es una instancia, ¿pero de qué clase es instancia? La respuesta es que **Object**, como todas las clases, es instancia de **Class**.

```
assertSame(Object.class.getClass(), Class.class);
```

¿Y su superclase? **Object**, al ser la raíz de toda la jerarquía que existe en Java no tiene superclase. Su superclase sería **null**.

```
assertNull(Object.class.getSuperclass());
```

La jerarquía de clases en Java es chata ya que todas ellas son instancias de una única clase: **Class**. Esto significa que todas tienen los mismos métodos. Recordemos que los métodos estáticos no son métodos pertenecientes a la clase, en el sentido que no son métodos

de instancia de los objetos clases. Hay otra forma de obtener el objeto que modela una clase en particular y es mediante el método estático de **Class**, **forName**. Este método permite, utilizando el nombre completo de una clase (incluido el paquete), cargar esa clase y devolverla.

```
assertSame(Class.forName("java.util.Date"), Date.class);
```

Esta forma se utiliza en casos muy puntuales y en general no es recomendable usarla, ya que podemos obtener errores en tiempo de ejecución debido a que esa clase no existe o su nombre está mal escrito.

```
@Test(expected=ClassNotFoundException.class)
public void testClassForName() throws ClassNotFoundException {
    Class.forName("clase.no.Existente");
}
```

Es necesario tener en cuenta que al ser objetos, cuando **debuggeamos** en Eclipse, es posible inspeccionar las instancias de clase del mismo modo en que lo haríamos con cualquier otro objeto.

Si queremos, podemos crear una nueva instancia de esta clase sin necesidad de llamar directamente a un constructor. Para realizar esta acción debemos proceder a enviar el mensaje **newInstance**. Debemos saber que este método utiliza el constructor por defecto (sin agregar argumentos) si se encuentra disponible.

```
StringBuilder builder = StringBuilder.class.newInstance();
```



NOMBRES DE LAS CLASES

Las clases en Java tienen nombres y tienen un paquete al que pertenecen. El nombre completo de una clase está formado por el nombre del paquete más el nombre de la clase, por ejemplo **java.lang.Object**. Las clases anidadas agregan su propio nombre al nombre completo de la clase contenedora, pero no utilizan el punto sino que usan el signo **$**, por ejemplo, **java.util.Map$Entry**.

Notemos que este método devuelve una instancia del tipo correcto, esto se debe a que **Class** es en realidad una clase genérica. Por ejemplo la clase **String** es del tipo **Class<String>**. De esta forma no necesitamos castear y el compilador se asegura de que los tipos estén correctos.

Métodos

Enfoquémonos ahora en ver cómo los métodos de nuestros objetos se representan como objetos. Una vez que estamos tratando con el objeto clase, estamos en el nivel meta del lenguaje y podemos consultar a la clase por sus distintos elementos. Dado que los métodos de una clase pueden ser tanto heredados como pertenecientes (declarados) a ella misma, hay varias formas de consultarlos. Primero podemos obtener todos los métodos de una clase utilizando el método **getMethods()**. Tengamos en cuenta que devuelve todos los métodos de instancia públicos, ya sean declarados en ella o heredados de sus superclases o de sus interfaces y superinterfaces.

```
Method [] metodos = String.class.getMethods();
```

Si sabemos cuál es el método que queremos obtener, sabemos su nombre y los tipos de los argumentos; podemos pedir por ese método directamente. Para esto se utiliza el método **getMethod**. Este aplica la siguiente lógica para buscar el método solicitado:

1) Realizar la búsqueda en la clase actual de los métodos públicos con el nombre que hemos especificado y que tengan exactamente los mismos tipos de argumentos solicitados.
2) Si se encuentra el método que estamos buscando, se devuelve. Si no, en el caso que la clase tenga superclase, se vuelve a 1 pero con la superclase como clase actual. Si no tiene superclase se arroja una excepción del tipo **NoSuchMethodException**.

```
Method toString = Object.class.getMethod("toString");
```

Ahora, si solamente queremos los métodos definidos en la propia clase y no nos interesan aquellos heredados, podemos reducir el espacio de consulta aplicando los métodos primos a los dos que ya vimos, **getDeclaredMethods** y **getDeclaredMethod**. Ambos funcionan de manera similar, pero se restringe la búsqueda solamente a la clase actual y de esta forma no se tienen en cuenta las superclases. Si estamos utilizando **getDeclaredMethods**.

```
Method [] metodos = String.class.getDeclaredMethods();
```

Y utilizando **getDeclaredMethod**.

```
Method equals = Date.class.getDeclaredMethod(
                    "equals", Object.class);
```

Es necesario tener en cuenta que la particularidad que tienen estos métodos es que permiten obtener aquellos que sean públicos, protegidos o privados, cosa que las otras versiones no permiten.

Una vez que obtenemos un objeto del tipo **Method** que representa a un método de instancia, podemos hacer varias cosas con él. Podemos obtener información sobre los argumentos que acepta, el tipo de objeto que devuelve como resultado, las excepciones que tiene declaradas para arrojar, su nombre y muchas otras cosas. También podemos invocarlo pasándole el objeto que vendría a ser el **this**, o sea el receptor.

De esta forma, si deseamos conocer los argumentos que son aceptados por el método en cuestión será necesario que hagamos uso de los mensajes **getParameterTypes**.



ENCAPSULAMIENTO

Si bien podemos tener acceso a los métodos privados de una clase utilizando **getDeclaredMethods**, esto no es una buena práctica, ya que estamos rompiendo el encapsulamiento. Siempre que utilizamos reflexión, deberíamos enfocarnos en los elementos públicos de una clase y no inmiscuirnos en sus asuntos íntimos. Además, tendríamos que tener mucho conocimiento sobre estos elementos para utilizarlos.

```
// conseguimos el método
Method compareTo = Date.class.getMethod("compareTo", Date.class);

// conseguimos los tipos de los parámetros
Class<?>[] parameters = compareTo.getParameterTypes();

// verificamos que sean correctos
assertEquals(parameters.length, 1);
assertSame(parameters[0], Date.class);
```

Los tipos de los argumentos siempre se pasan y se reciben en el mismo orden en el que están definidos en el método.

Si estamos lidiando con una clase genérica, con el genérico instanciado, obtendremos, al consultar los tipos de los argumentos, los tipos correctos y esperados. Por ejemplo, con **Date**, que implementa **Comparable<T>**, el método **compareTo** toma un **Date** como argumento. También podemos consultar los argumentos genéricos y estudiar la definición real, genérica o del método, usando **getGenericParameterTypes**.

```
// conseguimos el método
Method compareTo = Date.class.getMethod("compareTo", Date.class);
// conseguimos los tipos de los parámetros
Type[] genericParameters = compareTo.getGenericParameterTypes();

// verificamos que sean correctos
assertEquals(genericParameters.length, 1);
assertSame(genericParameters[0], Date.class);
```



SOBRE LOS PARÁMETROS

Cuando indicamos los parámetros al momento de buscar un determinado método, debemos pasarlos tal cual están declarados en la firma de este. No sirve, por ejemplo, pasar una subclase o subinterface, tiene que ser exacto. Esto es un problema cuando tenemos los argumentos y queremos encontrar el método que los acepta. En esos casos debemos buscar manualmente el método.

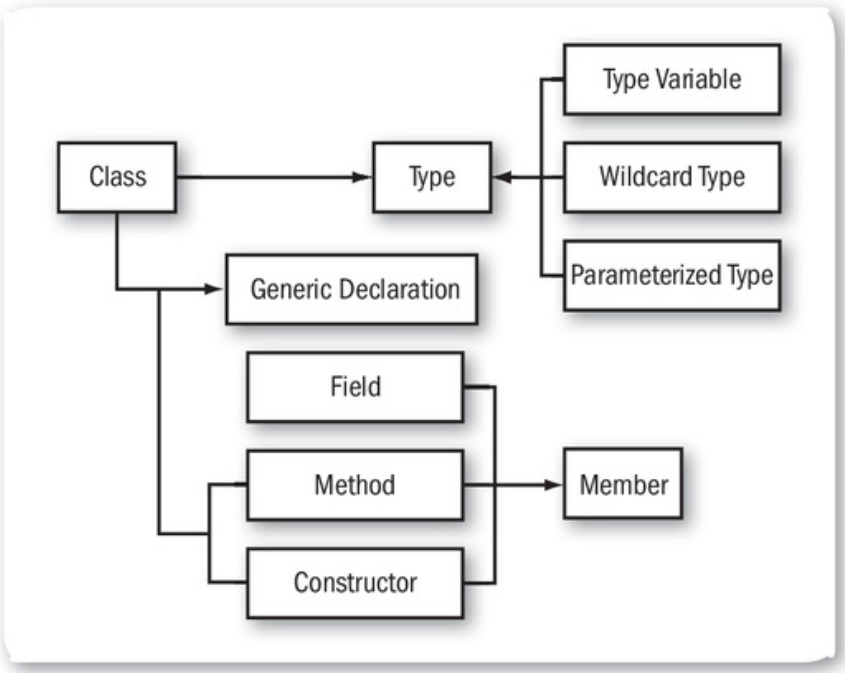


Figura 4. Esta imagen nos muestra la jerarquía de **Type**.

Muy similar al código normal, con la particularidad de que usamos **Type** en vez de **Class**. **Type** es una interfaz que tiene varias interfaces hijas para representar los distintos tipos de declaraciones con genéricos. Y en particular **Class** implementa **Type**. En este caso obtenemos la clase **Date** en el parámetro genérico. En cambio, si estamos analizando una clase genérica no instanciada, como el caso de **AbstractCollection**, entonces tenemos algunas diferencias. En principio, los tipos de los parámetros que están declarados como genéricos en **Object**.

```
// conseguimos el método
Method add = AbstractCollection.class.getMethod("add", Object.class);

// conseguimos los tipos de los parámetros
Class<?>[] parameters = add.getParameterTypes();

// verificamos que sean correctos
assertEquals(parameters.length, 1);
assertSame(parameters[0], Object.class);
```

Recordemos que si consultamos los parámetros genéricos, esta vez no obtendremos clases, como en el caso anterior, sino que obtendremos otro tipo de objeto. En este caso serán objetos del tipo **TypeVariable**. Un **TypeVariable** representa el uso de un tipo genérico definido ya sea en un método o en una clase.

```
// obtenemos el método
Method add = AbstractCollection.class.getMethod("add", Object.class);

// obtenemos los tipos de los parámetros
Type[] genericParameters = add.getGenericParameterTypes();

// los verificamos
assertEquals(genericParameters.length, 1);
assertTrue(genericParameters[0] instanceof TypeVariable);

// verificamos el tipo genérico
if(genericParameters[0] instanceof TypeVariable) {
    TypeVariable<Class<?>> type = (TypeVariable<Class<?>>)
    genericParameters[0]; assertSame(type.getBounds()[0], Object.class);
}
```

La invocación de un método sobre un determinado objeto se realiza mediante el mensaje **invoke**. El método **invoke** espera como primer argumento al teórico receptor del mensaje, al objeto que será el **this** del método. Luego acepta los argumentos del método y devuelve el resultado de la ejecución o **null** si el método devolvía **void**.

Si el método a invocar es un método estático, entonces no es necesario pasar el this ya que se ignora ese argumento completamente.



CGLIB

Una librería para generar código **bytecode** en Java es **CGLib**. Esta librería la podemos encontrar en **http://cglib.sourceforge.net** y nos permite extender clases e interfaces en tiempo de ejecución. Esta librería es muy utilizada por varios frameworks, como por ejemplo **Hibernate** y **Spring**. Ambos la usa para generar proxies de clases, uno para las persistentes y el otro para implementar **AOP**.

```
// conseguimos el método
Method toString = Integer.class.getMethod("toString");

// invocamos el método en el objeto 4
assertEquals(toString.invoke(Integer.valueOf(4)), "4");
```

```
c:\>javap -c java.util.Map
Compiled from "Map.java"
public interface java.util.Map{
public abstract int size();

public abstract boolean isEmpty();

public abstract boolean containsKey(java.lang.Object);

public abstract boolean containsValue(java.lang.Object);

public abstract java.lang.Object get(java.lang.Object);

public abstract java.lang.Object put(java.lang.Object, java.lang.Object)

public abstract java.lang.Object remove(java.lang.Object);

public abstract void putAll(java.util.Map);

public abstract void clear();

public abstract java.util.Set keySet();

public abstract java.util.Collection values();

public abstract java.util.Set entrySet();

public abstract boolean equals(java.lang.Object);

public abstract int hashCode();

}
```

Figura 5. Podemos darnos cuenta de que **javap** nos muestra que la información de los genéricos no se encuentra disponible.

INSPECCIONANDO OBJETOS EN ECLIPSE

Es interesante tener en cuenta que cuando estamos **debuggeando** código en **Eclipse** podemos inspeccionar la estructura interna de los objetos que están en los distintos **stackframes** en la vista de **Variables**. Además de esto, es posible observar los distintos métodos, tanto públicos como privados y protegidos, en la vista de **Outline**. De esta forma, nos damos cuenta de que estas dos vistas son muy útiles para investigarlos y ver qué es lo que ocurre con nuestros objetos. Además en la vista de **Variables** podemos modificar los atributos de los objetos por algunos otros en los casos de tipos básicos.

Constructores

Los constructores son, es cierta forma, parecidos a los métodos, con la particularidad que al ejecutarse se obtiene una nueva instancia. La consulta de los tipos de parámetros aceptados, como de las excepciones chequeadas que pueden arrojar, y la de los genéricos utilizados se realiza de igual manera que con los métodos. La obtención de los constructores se logra mediante los métodos **getConstructor**, **getConstructors**, **getDeclaredConstructor** y **getDeclaredConstructors** de la clase. La única diferencia es que no es necesario pasar un nombre ya que los constructores no lo tienen. Al invocar un constructor para crear una nueva instancia, obviamente no es necesario pasar como parámetro el **this**, como en el caso de los métodos, ya que este será la instancia recientemente creada.

Al invocar un constructor, debemos tomar los mismos recaudos que cuando invocamos un método. En primer lugar será necesario que controlemos los argumentos, los tipos primitivos y también cada una de las excepciones que se podrían lanzar. de esta forma podremos estar seguros de que todo se ejecutará según lo planificamos.

```
// conseguimos el constructor sin parámetros
Constructor<HashSet> ctr = HashSet.class.getConstructor();

// creamos una nueva instancia
HashSet set = ctr.newInstance();

// y la verificamos
assertTrue(set.isEmpty());
```



REFLEXIÓN Y LOS GENÉRICOS

Dado que los genéricos en Java están implementados principalmente en el compilador, no es posible, por ejemplo, crear instancias que especifiquen un genérico. No podemos, por reflexión, utilizar el constructor de **ArrayList** para crear una lista de **String**s, ya que no son parte del mismo tipo. Simplemente podremos crear un **ArrayList** sin ningún tipo específico. Generalmente esto no es un problema.

Atributos

Al igual que con los métodos y los constructores, acceder a los atributos de una clase es sencillo y sigue la misma estructura. **Class** provee los siguientes métodos para obtener los atributos, tanto los declarados en la propia clase como los heredados, **getFields**, **getField**, **getDeclaredFields** y **getDeclaredField**. Al igual que en el caso de los métodos y de los constructores, **getField** y **getFields** permiten obtener los atributos públicos tanto propios como heredados; y **getDeclaredField** y **getDeclaredFields** pueden obtener los protegidos y privados aunque solamente los propios. Cuando buscamos un campo (o atributo) en particular, solamente necesitamos especificar su nombre.

```
public class Contenedor<T> {
    private T valor;

    public void setValor(T valor) {
        this.valor = valor;
    }

    public T getValor() {
        return valor;
    }

}

// creamos el objeto
Contenedor<String> texto = new Contenedor<String>();
```



MIEMBROS SINTÉTICOS

Tengamos en cuenta que si investigamos un poco los protocolos que corresponden a los métodos, constructores y atributos, veremos que implementan un método llamado **isSynthetic()**. Este indica si el elemento fue introducido por el compilador. Ejemplo de esto es el **this** de la instancia de la clase contenedora cuando estamos en una clase anidada no estática.

```
// accedemos al campo
Field valor = texto.getClass().getDeclaredField("valor");

// como es privado, lo habilitamos
valor.setAccessible(true);

// chequeamos el tipo (Object por el type erasure)
assertSame(valor.getType(), String.class);

// verificamos que está vacío
assertNull(valor.get(texto));

// le ponemos un valor
valor.set(texto, "hola");

// y lo verificamos
assertEquals(texto.getValor(), "hola");
```

Recordemos que los atributos deberían ser privados y por lo tanto no deberíamos interferir con ellos y los atributos públicos deberían no usarse nunca. Por lo tanto acceder a los atributos por reflexión es raro. Además, si normalmente no tenemos visibilidad del atributo, debemos hacer uso del método **setAccessible** para poder leerlo y modificarlo, si no, obtendremos una excepción. La clase **Field**, que representa a los atributos de una clase, ofrece métodos para leer y escribir su valor. Ofrece versiones específicas para los tipos de datos primitivos y luego una versión para **Object**. Cuando queremos operar sobre el campo de un objeto particular, debemos pasarlo como parámetro.



DECOMPILADOR JAVA

Si nos dirigimos a **http://java.decompiler.free.fr** encontraremos una pequeña aplicación que sirve para descompilar las clases Java binarias y ver así su código fuente. Este utilitario es muy práctico cuando queremos saber cómo funciona cierta clase y no disponemos del código fuente. La aplicación acepta archivos .class directamente y los archivos .JAR (archivos .ZIP renombrados con clases adentro).

Modificadores

Las clases, los atributos, los constructores y los métodos tienen, como sabemos, modificadores. No solamente tenemos los modificadores de visibilidad **private**, sino también tenemos algunos otros, como **final**, **abstract**, **synchronized**. Si bien es posible acceder a los modificadores de los elementos mediante reflexión, lamentablemente, el diseño deja bastante que desear ya que es muy primitivo. Los modificadores no están modelados, sino que cada uno está representado por un número, y el conjunto de los modificadores de un elemento está dado por una máscara de bits. Para solventar esto, existe la clase **Modifier** que ofrece algunos servicios que facilitan la manipulación de los modificadores. Para obtener los modificadores de un elemento, utilizamos el método **getModifiers** que devuelve un **int** (mascara de bits). Podemos notar lo engorroso que puede resultar el uso de esta API debido al diseño primitivo y poco orientado a objetos. Por esta razón puede traernos algunas complicaciones que debemos tener en cuenta antes de usarla.

> LOS MODIFICADORES NO ESTÁN MODELADOS, SINO REPRESENTADOS POR UN NÚMERO

```
// obtenemos los modificadores del método hashCode
int modificadores = Object.class.getMethod("hashCode").getModifiers();

// verificamos si es público
assertTrue(Modifier.isPublic(modificadores));

// verificamos si es nativo
assertTrue(Modifier.isNative(modificadores));

// verificamos si no es final
assertFalse(Modifier.isFinal(modificadores));

// verificamos si es de instancia
assertFalse(Modifier.isStatic(modificadores));
```

APIs.

Interfaces

Para la reflexión, debemos tener en cuenta que las interfaces son clases abstractas decoradas con el modificado de **interface**. Por lo tanto todo lo que vimos para las clases aplica para las interfaces. Al igual que con las clases abstractas, vamos a obtener una excepción si tratamos de instanciar un objeto con el objeto clase de la interfaz.

```
// obtenemos los modificadores de la interfaz
int modificadores = Set.class.getModifiers();

// verificamos que es pública
assertTrue(Modifier.isPublic(modificadores));

// verificamos que es abstracta
assertTrue(Modifier.isAbstract(modificadores));

// verificamos que es una interfaz
assertTrue(Modifier.isInterface(modificadores));
```

Clases anidadas

Ya vimos cómo, por reflexión, podíamos acceder a la estructura de las clases, los atributos, los constructores y los métodos. También, que las interfaces se ven como clases abstractas. Ahora vamos a aprender cómo inspeccionar las clases anidadas. Para comenzar, tenemos el método **getClasses**, que retorna todas las clases e interfaces públicas declaradas en esta clase o en clases superiores.

```
assertSame(
    Map.class.getClasses()[0], Map.Entry.class);
```

Luego, nos encontramos con el método **getDeclaredClasses**, que devuelve un array con las clases e interfaces (todas, no solo las

públicas) declaradas en esta clase. No se proveen métodos para acceder directamente por nombre a una clase anidada. Ahora, si tenemos un objeto de una clase que es anidada, entonces podemos conocer el contexto en que esta fue definida. Utilizamos para esto el método **getDeclaringClass**, que devuelve la clase donde está definida la clase anidada. Las clases anónimas (y las locales a métodos) no tienen una clase donde están definidas, pero sí tienen una que las envuelve, esto lo obtenemos mediante el método **getEnclosingClass**.

```
// creamos una clase anónima
Class<?> type = new Comparable<Integer>() {
    @Override
    public int compareTo(Integer arg0) {
        return -1;
    }
}.getClass();

// verificamos que no tiene clase declarante
assertNull(type.getDeclaringClass());

// pero sí tiene una clase que la engloba
assertSame(type.getEnclosingClass(),
this.getClass());
```

Es importante recordar que si queremos saber cuál es el método o constructor donde se declara una determinada clase anónima o local, podemos utilizar los métodos de consulta siguientes: **getEnclosingMethod** y **getEnclosingConstructor**.



SOBRE EL DISEÑO DE MODIFIER

Es fácil ver que los modificadores no están modelados. La **API** que trata sobre los modificadores es muy primitiva y debería actualizarse al utilizar las enumeraciones que tiene Java y al hacer uso del **EnumSet**. En la forma actual, solo son números enteros, y necesitamos de **Modifier** y sus métodos estáticos para lidiar con ellos. Lamentablemente Java tarda en actualizar sus propias API.

```
@Test
public void testMemberClass() throws
        ClassNotFoundException,
        SecurityException,
        NoSuchMethodException {
  // creamos una clase anónima
  Class<?> type = new Comparable<Integer>() {
    @Override
public int compareTo(Integer arg0) {
  return -1;
}
  }.getClass();

  // verificamos que sea el mismo método
  assertEquals(
    this.getClass().
      getDeclaredMethod("testMemberClass"),
    type.getEnclosingMethod());
}
```



Métodos y atributos estáticos

Es necesario señalar que para que podamos acceder por reflexión a los métodos y atributos clasificados como estáticos, no tenemos que hacer nada especial, simplemente usar lo que hemos aprendido hasta ahora. Aunque será necesario que tengamos en cuenta unas pequeñas diferencias a la hora de trabajar con ellos.

```
// obtenemos el método min
Method min = Math.class.
        getMethod("min", int.class, int.class);
// verificamos que funciona
assertEquals(min.invoke(null, 3, 4), 3);
```

Una de las diferencias entre los elementos de instancia y los estáticos durante la reflexión es que tienen como modificador a **static**. Además ignoran el objeto pasado como **this**. En los métodos se ignora usando el método **invoke**, y en los atributos, cuando queremos leerlos o escribirlos, usando **get** y **set**. Incluso podemos pasar como parámetro a **null**.

Arrays

El trato que reciben los array es especial, ya que son objetos manejados por la máquina virtual y el lenguaje. Para acceder a los array usando reflexión debemos tener algunas consideraciones. Para discriminar entre las clases comunes y los array utilizamos el método **isArray()** provisto en **Class**. Las clases de los array tienen nombres particulares. Primero, tienen tantos caracteres de este tipo **[** como dimensiones tiene el array. Luego, dependiendo del tipo de dato del array, utiliza una codificación distinta. Finaliza con este carácter **;**.

ELEMENTOS Y SU CODIFICACIÓN

TIPO DE ELEMENTO	CODIFICACIÓN
boolean	Z
byte	B
char	C
objeto	L <nombre de la clase o interface>
doublé	D
float	F
int	i
long	J
short	S

Tabla 1. Aquí vemos los elementos y su respectiva codificación.

```
// creamos una array
String [][] array = new String [][] {};

// verificamos que la clase es un array
assertTrue(array.getClass().isArray());

// verificamos el nombre
assertEquals(
        array.getClass().getName(),
        "[[Ljava.lang.String;");
```

Debemos saber que también contamos con una clase auxiliar llamada **Array** la cual se encarga de brindar una gran ayuda para trabajar con array y reflexión. Por ejemplo podemos acceder a creación de nuevos array utilizando el método denominado **newInstance**. Este método espera el tipo de los elementos y las dimensiones.

```
// creamos un array
String [] a = (String[])
        Array.newInstance(String.class, 3);

// verificamos la longitud
assertEquals(3, a.length);

// verificamos que estén en null los elementos
assertArrayEquals(
        new String [] {null, null, null}, a);
```



ARRAYS Y LAS COLECCIONES

Es una lástima que los array sean parte del lenguaje (y del compilador y la máquina virtual) y no sean en sí un tipo de colección. Primero podríamos tratarlos polimórficamente con una colección. Segundo, no necesitarían tanto soporte del lenguaje, el compilador y la máquina virtual. Tercero, no sería necesario tener clases y protocolo para tratarlos por reflexión. La performance tampoco sería un problema.

Además, **Array** ofrece métodos para leer y escribir valores en los array, así como también obtener su la longitud. Los métodos para acceder a los elementos están discriminados por tipos primitivos y luego por uno general para objetos.

```
assertEquals(Array.getLength(new int [] {4, 3}), 2);

assertEquals(
        Array.getLong(new long [] { 2L }, 1),
        2
);
```

Enumeraciones

Como vimos en el capítulo dedicado a ellas, las enumeraciones son clases que definen ciertos atributos estáticos donde cada referencia es una instancia de la misma clase (o una subclase anónima). Por lo tanto podemos utilizar las mismas herramientas que vimos para inspeccionar y manipular clases y atributos mediante reflexión. Además existe cierto protocolo específico para cuando se da el caso de las enumeraciones.

Empezaremos viendo cómo determinamos que una clase es en realidad una enumeración. Para ello usamos el método **isEnum** perteneciente a **Class**, de la siguiente forma:

```
// declaramos una enumeración
enum TEST {
```



VELOCIDAD DE LA REFLEXIÓN

Tenemos que ser cuidadosos a la hora de usar estas herramientas para inspeccionar la estructura de los objetos. Ejecutar un método por reflexión es mucho más lento que hacerlo directamente, así como también lo es el acceso a los atributos. Entonces no debemos abusar, ya que si lo hacemos, la velocidad de nuestra aplicación caerá rápidamente. Si es necesario podemos utilizar **caching** para ganar algo de velocidad.

```
        A, B, C, D, E
}

// verificamos que la clase está marcada como tal
assertTrue(TEST.class.isEnum());
```

```
c:\>javap -c red.user.java.Palo
Compiled from "Palo.java"
public final class red.user.java.Palo extends java.lang.Enum{
public static final red.user.java.Palo PICA;

public static final red.user.java.Palo CORAZON;

public static final red.user.java.Palo DIAMANTE;

public static final red.user.java.Palo TREBOL;

static {};
  Code:
    0:   new     #1; //class red/user/java/Palo
    3:   dup
    4:   ldc     #17; //String PICA
    6:   iconst_0
    7:   ldc     #18; //String ?
    9:   invokespecial   #20; //Method "<init>":(Ljava/lang/String;ILjava/lang
ring;)V
   12:   putstatic       #24; //Field PICA:Lred/user/java/Palo;
   15:   new     #1; //class red/user/java/Palo
   18:   dup
   19:   ldc     #26; //String CORAZON
   21:   iconst_1
   22:   ldc     #27; //String ?
   24:   invokespecial   #20; //Method "<init>":(Ljava/lang/String;ILjava/lang
ring;)V
   27:   putstatic       #29; //Field CORAZON:Lred/user/java/Palo;
   30:   new     #1; //class red/user/java/Palo
   33:   dup
   34:   ldc     #31; //String DIAMANTE
   36:   iconst_2
   37:   ldc     #32; //String ?
   39:   invokespecial   #20; //Method "<init>":(Ljava/lang/String;ILjava/lang
ring;)V
```

Figura 10. En esta imagen vemos la salida de ejecutar **javap** donde las enumeraciones son campos estáticos de la clase.

Cuando queríamos saber todos los elementos de una enumeración utilizábamos el mensaje **values()**. Ahora, para obtener el mismo resultado en el código que utilizamos, le enviamos el mensaje **getEnumConstants()** a la clase de la enumeración. Este método nos devuelve un array con los distintos elementos.

```
assertArrayEquals(
        TEST.values(), TEST.class.getEnumConstants());
```

Ahora, al ser clases las enumeraciones, podemos inspeccionar sus elementos utilizando reflexión. Encontraremos que hay varios métodos y constructores en las enumeraciones, tengamos en cuenta que estos son generados por el compilador automáticamente.

Ahora vamos a obtener uno de los elementos como atributo.

```
// obtenemos el atributo para el elemento SECONDS
Field attr = TimeUnit.class.getField("SECONDS");

// verificamos que esté marcado como enum
assertTrue(attr.isEnumConstant());

// recuperamos el valor del atributo
TimeUnit seconds = (TimeUnit) attr.get(null);

// y lo comparamos con el elemento directamente
assertSame(seconds, TimeUnit.SECONDS);
```

Proxy y generación de código

Se conoce como **proxy** a un objeto que está representando a otro y restringe su acceso. Proxy es uno de los patrones de diseño más conocidos. Se lo utiliza, por ejemplo, para cuando no queremos tener un objeto gigante en memoria, de modo que creamos un proxy y



FORMA DE LAS ENUMERACIONES

Cuando accedemos por reflexión a las enumeraciones y las inspeccionamos, podemos ver que en realidad son un truco del lenguaje y que el compilador genera clases y atributos para ellas. Son elementos para el lenguaje pero para la máquina virtual son solamente clases comunes y corrientes con atributos y métodos. Esto es un buen ejemplo de cómo abstraer patrones de código.

hacemos que solamente se cargue el otro objeto cuando alguien hace uso del proxy. Para el cliente, el proxy y el otro objeto (el proxeado) son indistinguibles, para que esto suceda el proxy debe ser polimórfico con el objeto que desea proxear.

> JAVA OFRECE LA POSIBILIDAD DE CREAR PROXIES EN FORMA DINÁMICA

En Java esto significa que debe heredar de una clase o implementar una interfaz funcional a ambos. En Java existen infinidad de frameworks que permiten generar dinámicamente proxies de objetos en tiempo de ejecución, de esta forma podemos simular una modificación en el comportamiento de dicho objeto. Si bien el patrón establece que un proxy sirve para controlar el acceso, es muy común que se agrupe bajo esta denominación a otros patrones muy parecidos (el **decorator**, por ejemplo, que sirve para agregar comportamiento) los que son, en definitiva, **wrappers** (envoltorios) de los otros objetos.

Java ofrece, en su librería base, una forma de crear proxies dinámicamente. Estos proxies son clases generadas en tiempo de ejecución que implementan ciertas interfaces. En definitiva, no deja crear de forma reflexiva clases que implementen varias interfaces y especifiquen su comportamiento, sin que estas clases existan en el código fuente. Lamentablemente existe la limitación de que no se pueden proxear clases con este mecanismo. Los frameworks utilizan otras técnicas para solventar este problema. Para continuar vamos a ver un ejemplo sencillo de creación de proxy. Podemos comenzar creando un par de interfaces para poder proxear.

```
interface ConNombre {
    String nombre();
}

interface ConApellido {
    String apellido();
}
```

Ahora, para crear proxies, Java nos ofrece la clase utilitaria **Proxy**. Esta brinda dos métodos de creación, uno que permite originar una clase que

implementa las interfaces especificadas, y otro que directamente crea una instancia de una clase con cierto comportamiento que las implementa. Utilizaremos el segundo, que es más directo. La firma del método es.

```
static Object newProxyInstance(
    ClassLoader loader,
    Class<?> [] interfaces,
    InvocationHandler handler
);
```

Estudiemos un momento los parámetros que requiere el método. El primero, es un **ClassLoader**, los **ClassLoader**s son objetos que se ocupan de cargar las clases en la máquina virtual. Por otra parte, el segundo, son las interfaces que queremos que implemente la clase que se va a crear. El tercero, es un objeto que se encarga de darle el comportamiento a la nueva instancia. **InvocationHandler** define un único método que es invocado cada vez que le envía un mensaje al proxy. Cuando se quiere ejecutar un método en el proxy, se llama al método **invoke** del **InvocationHandler** pasado. El método **invoke** recibe el proxy (receptor del mensaje), el objeto **Method** que representa al método que se va a ejecutar y luego los argumentos pasados. A continuación podemos ver nuestro **InvocationHandler** de ejemplo.

```
class ConNombreYApellido implements
    InvocationHandler {

    private final String nombre;
```



MANIPULADORES DE BYTECODE

Como ya vimos, estamos limitados con respecto a la modificación del comportamiento de un método o para proxear clases. Para superar estas limitaciones existen varias librerías de manipulación de **bytecode**. Estas librerías permiten leer y escribir el binario de un método o clase en tiempo de ejecución. Esto, en conjunto con los **ClassLoader**s, nos permite construir las clases que necesitamos.

```
    private final String apellido;

    public ConNombreYApellido(
        String nombre, String apellido) {
            super();
            this.nombre = nombre;
            this.apellido = apellido;
}
@Override
public Object invoke(
        Object proxy,
        Method method,
        Object[] arguments)
throws Throwable {
            String name = method.getName();
            if(name.equals("nombre")) {
                return nombre();
            }
            if(name.equals("apellido")) {
                return apellido();
             }
             return null;
    }

    public String nombre() {
            return nombre;
    }

    public String apellido() {
            return apellido;
    }
}
```

¿Y el **ClassLoader**? El **ClassLoader** lo obtendremos de la propia clase del test unitario que usaremos para probar esto.

Debemos tener en cuenta de que toda clase conoce y tiene una referencia al **ClassLoader** que la cargó en forma original.

```
// creamos el proxy
Object proxy = Proxy.newProxyInstance(
    // obtenemos el ClassLoader
    this.getClass().getClassLoader(),
    // especificamos las interfaces
    new Class<?>[] {
            ConNombre.class,
       ConApellido.class },

    // y ahora el InvocationHandler
    new ConNombreYApellido("Juan", "De Los Palotes"));

// verificamos que el proxy implementa las interfaces
assertTrue(proxy instanceof ConNombre);
assertTrue(proxy instanceof ConApellido);

// verificamos la implementación
assertEquals("Juan", ((ConNombre)proxy).nombre());
assertEquals("De Los Palotes", ((ConApellido)proxy).apellido());
```

Gracias a **Proxy** hemos podido crear un objeto de una clase que no existe en el código fuente y que se generó en tiempo de ejecución. Este tipo de código es muy útil para aumentar el comportamiento de los objetos como para implementar acciones que se realizan antes o después, o en lugar de otras. Por ejemplo podemos aplicar las validaciones que hemos creado en un capitulo anterior usando esta técnica. Solamente tenemos que crear un **InvocationHandler** que haga las validaciones pertinentes y luego llame al método en el objeto original.



ARTÍCULOS SOBRE CLASSLOADERS

En el sitio **JavaWorld** ubicado en **www.javaworld.com** podemos encontrar información sobre el funcionamiento de los **ClassLoader**s, así como también de su relación con la máquina virtual. Algunos artículos tratan el tema de la creación de **ClassLoader**s propios, y contienen código de ejemplo que nos permitirá iniciar nuestros pasos en esta área. El sitio está íntegramente en inglés.

Hay muchos otros usos y frameworks que facilitan esta tarea que es comúnmente conocida como **AOP** (**Aspect Oriented Programming** o **Programación Orientada por Aspecto**).

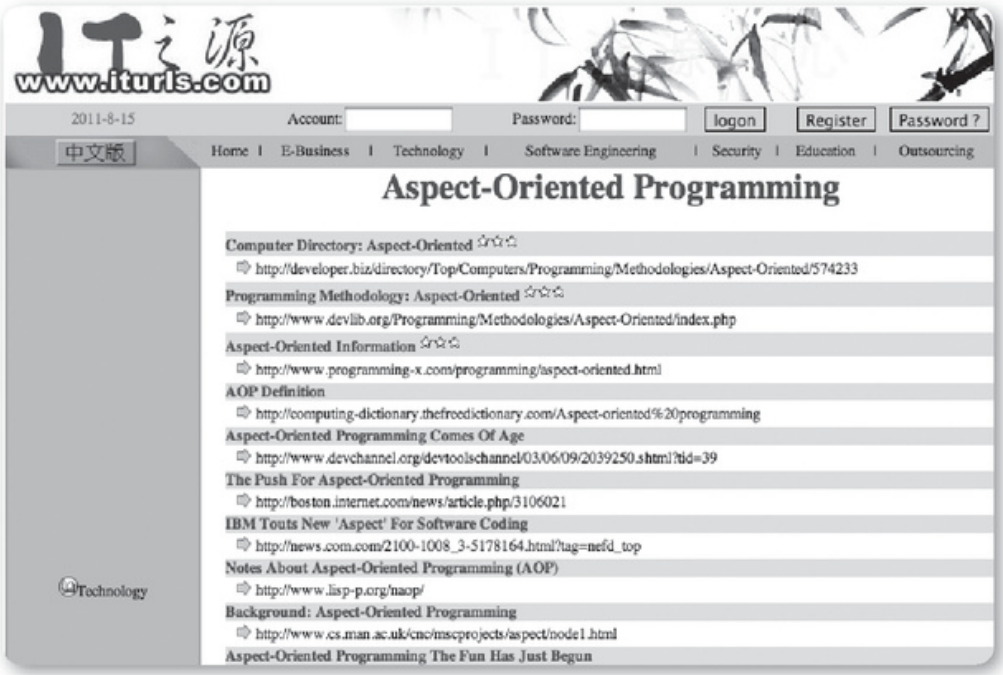


Figura 11. En la dirección **`web www.iturls.com/english/techhotspot/TH_e.asp`** encontramos una serie de interesantes enlaces relacionados con la programación Orientada por Aspecto.

Los ClassLoaders

Los **ClassLoader**s son objetos que se ocupan de cargar las clases del sistema a medida que se necesitan. La máquina virtual utiliza algunos **ClassLoader**s por defecto para cargar las clases básicas desde el sistema de archivos. Normalmente una aplicación no debe ocuparse de este tema y mucho menos crear un nuevo **ClassLoader**. Las que sí lo hacen los crean para poder cargar las clases desde distintos lugares como la red (como las **Applet** de Java), una archivo comprimido (como los servidores de aplicaciones Java) o, por cuestiones de seguridad, para restringir el uso de ciertas clases (como el **Google Application Engine**).



Figura 12. En la dirección **http://code.google.com/intl/es-ES/appengine** encontramos el sitio web de Google Application Engine.

Los **ClassLoader**s conforman una cadena donde las clases se buscan primero utilizando los **ClassLoader**s padres y se continúa bajando en la cadena hasta que alguna puede resolverla.

RESUMEN

Hemos aprendido en esta sección sobre la reflexión y cómo Java nos provee herramientas para inspeccionar la estructura de nuestros objetos en tiempo de ejecución. En cierta medida también nos ofrece la posibilidad de modificarlos. Vimos como en Java, las clases, los atributos, los métodos, los constructores y los otros elementos del lenguaje están representados por objetos con los que podemos interactuar. Esta gran funcionalidad permite escribir programas en Java que manipulen otros programas en Java o incluso a ellos mismos. También vimos cómo generar clases que utilizan proxy en tiempo de ejecución. Finalmente aprendimos como se cargan las clases en la máquina virtual.

Siguientes pasos

En este apartado trataremos de presentar, brevemente, algunos temas interesantes que complementarán nuestro conocimiento, no solo de Java, sino de la programación en general. Además tendremos un anticipo de cómo es el desarrollo de aplicaciones web en Java.

- Temas para seguir estudiando.....280
- Desarrollo de aplicaciones para la consola de comandos.....296
- Desarrollo de aplicaciones de escritorio.....298
- Desarrollando aplicaciones para dispositivos móviles.....301
- Desarrollando aplicaciones web.....303
- Otros lenguajes para la JVM.....307
- Resumen.....308

Temas para seguir estudiando

Primero mencionaremos algunos temas que complementarán lo estudiado a lo largo del libro. Algunos tratan exclusivamente sobre teoría de programación, aunque otros están referidos a una determinada tecnología. El lector deberá investigar estos temas él mismo y en la medida que pueda.

Estructuras de datos y otras colecciones

En la sección dedicada a las colecciones, vimos que Java nos ofrece las listas (**List**), los conjuntos (**Set**) y los diccionarios (**Map**). Además de esto, ofrece otras colecciones muy útiles de conocer.

Primero tenemos la interfaz **Queue<E>**, que modela un tipo muy simple de colección, donde podemos poner elementos en un extremo de ella y sacarlos por otro. No podemos interactuar con los elementos que están en el medio, solamente con los extremos. Hay dos grandes distinciones, primero están las **colas** o **queues** donde agregamos elementos al final y los sacamos por delante, de tal forma que el primero en entrar es el primero en salir. Esto se conoce comúnmente como **FIFO** o **First In, First Out**. Una analogía clara sería una cola en un banco, donde van llegando clientes y se los va atendiendo en el orden de llegada. Luego están las **pilas** o **stacks** donde agregamos elementos por el final (o tope) y sacamos también por el final (tope). Aquí el último elemento en entrar es el primero es salir, esto se conoce como **LIFO** o **Last In, First Out**.



WICKET

El sitio **http://wicket.apache.org** es el punto de entrada al mundo de **Wicket**. Este es un framework de desarrollo web orientado a componentes con estado. Esto significa que todo lo que haga el usuario queda en el servidor sin necesidad de que el programador lo tenga que hacer explícitamente. Esto permite que cierto tipo de aplicaciones sean sencillas de programar y contengan menos errores.

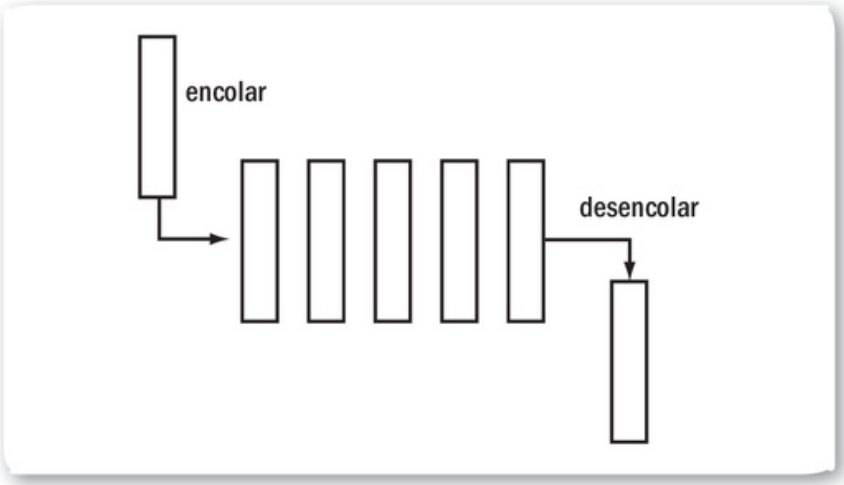


Figura 1. En esta imagen podmeos ver un diagrama que representa cómo funciona una cola.

Un ejemplo de esto sería una pila de platos donde vamos apilando platos y cuando queremos uno sacamos el de más arriba. En Java, la interfaz **Queue** engloba estas dos ideas y otras más (como las colas de prioridades, donde el primero en salir es el más importante) y existen varias clases que la implementan y algunas otras interfaces que la extienden. Para nombrar algunas, que el lector deberá investigar por su cuenta, tenemos **Dequeue**, **ArrayQueue**, **LinkedList** y **PriorityQueue**.

> EL ORDEN ESTA DADO POR LOS PROPIOS OBJETOS O MEDIANTE COMPARATOR

Otras colecciones por conocer que Java ofrece son los conjuntos y diccionarios ordenados y las extensiones navegables de estos. Los primeros mantienen a los elementos ordenados (según la clave en el caso del diccionario) y luego permiten recorrerlos en ese orden. El orden esta dado por los propios objetos (si implementan **Comparable**) y sino mediante un objeto **Comparator**. Los segundos son extensiones que ofrecen métodos para facilitar la navegación de los elementos que se encuentran contenidos.

Otras estructuras de datos que todo programador debe conocer, y que Java no ofrece por defecto, son los grafos y los árboles. Ambos representan elementos conectados por alguna relación (los árboles son casos especiales de los grafos, aunque en general se los trate por

separado). Los árboles sirven para representar muchas cosas; por ejemplo, una jerarquía de clases es, en realidad, una estructura arbórea.

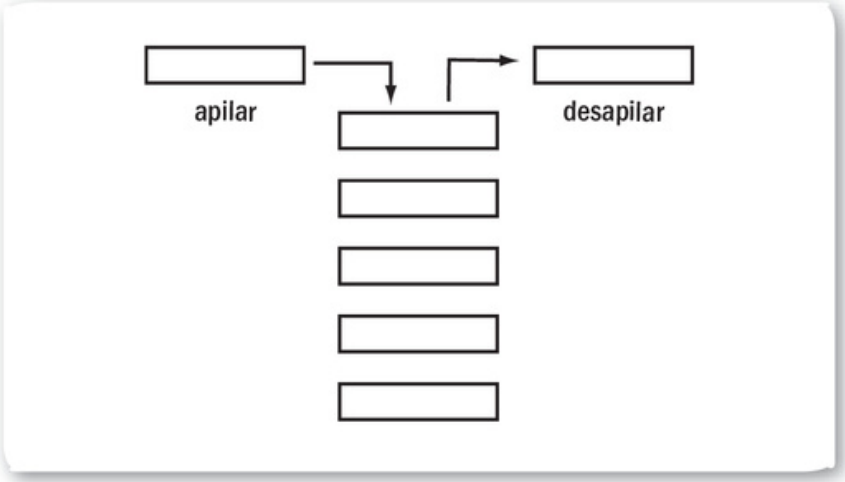


Figura 2. En esta imagen podemos ver un diagrama que representa el funcionamiento de una pila.

Existen infinidad de algoritmos que operan sobre estas estructuras y permiten solucionar una amplia variedad de problemas.

Java provee algunas otras implementaciones de las colecciones ya vistas, que el lector deberá investigar para tener en su repertorio.

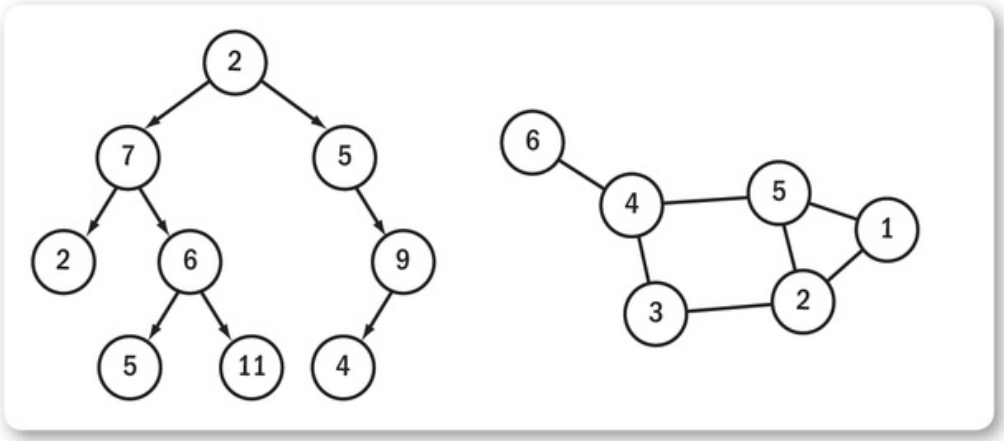


Figura 3. Aquí podemos ver dos diagramas, los cuales corresponden a un árbol binario y un grafo. Notar que un árbol es un grafo.

Algoritmos

Si bien los algoritmos están relacionados con el paradigma estructural, debemos conocerlos, ya que ofrecen una amplia gama de resoluciones a problemas. Nuestro objetivo será luego traducir estos algoritmos al paradigma de objetos. Algunos de los algoritmos más importantes que debemos conocer son aquellos utilizados para ordenar y buscar los elementos de las colecciones. Por nombrar algunos de los más conocidos tenemos, **búsqueda binaria** (para buscar elementos en una lista ordenada), y **quicksort**, **mergesort** y el **heapsort** (también para ordenar). Los algoritmos de búsqueda utilizando árboles binarios también son un conocimiento necesario, así como también lo son los algoritmos **breadth first search** y **deep first search** para recorrer grafos, o los de camino más corto entre dos elementos de un grafo (**Dijkstra**).

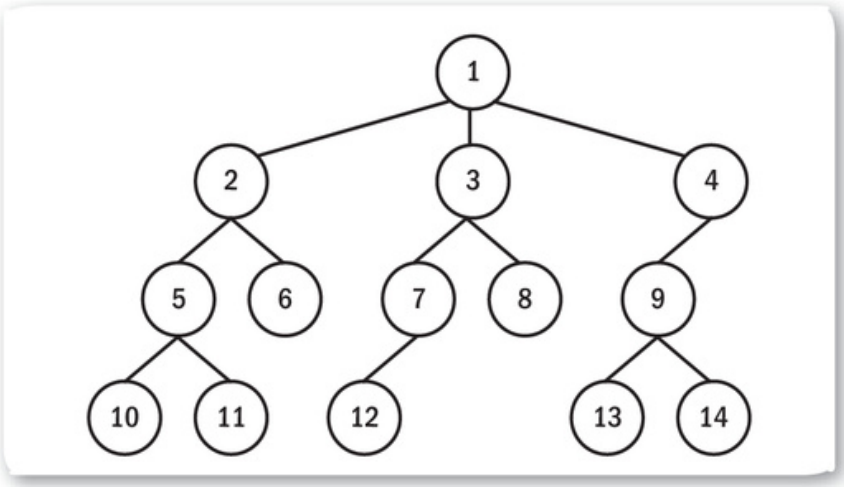


Figura 4. Diagrama del orden en que se recorren los nodos de un grafo usando el método **`breadth first search`**.



NO REINVENTAR LA RUEDA

Si nos enfrentamos ante una situación donde necesitamos de cierta estructura de datos determinada y Java no nos ofrece una implementación por defecto, debemos indagar en alguna librería que sí lo haga. No debemos lanzarnos sin meditar en la difícil tarea de conseguir implementar correcta y eficientemente tales estructuras de datos. Pensemos que siempre alguien necesitó esta solución antes que nosotros.

Todos estos algoritmos operan sobre las estructuras de datos y sobre las colecciones vistas y recomendadas en la sección anterior.

Es útil conocer sobre los órdenes de complejidad de los algoritmos para poder reconocer cuál es el más rápido de todos los posibles.

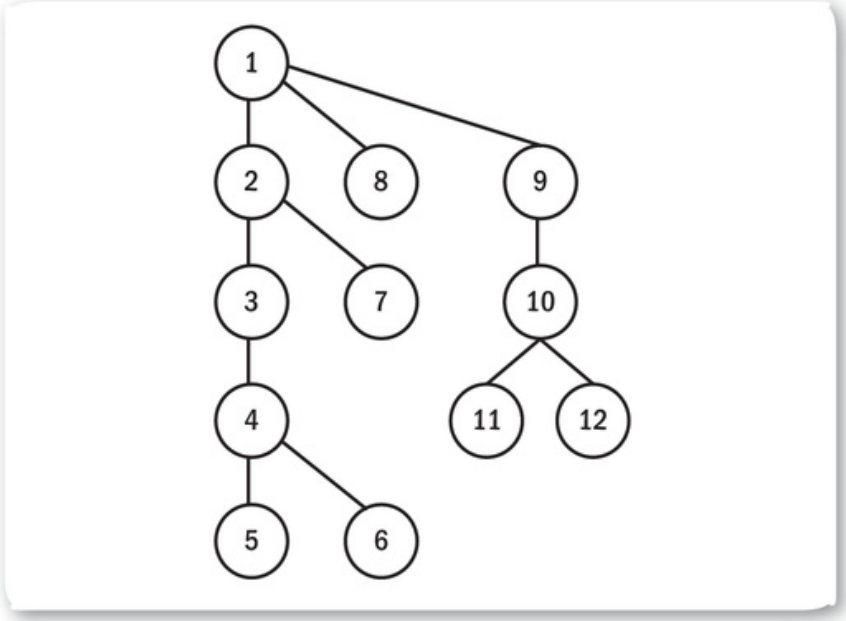


Figura 5. Diagrama del orden en que se recorren los nodos de un grafo usando el método **`depth first search`**.

Además de conocer ciertos algoritmos muy útiles, no está de más también familiarizarse con las distintas técnicas que existen para atacar un problema de forma algorítmica (por más que choque con el paradigma de objetos). Mencionemos algunas técnicas para algoritmos. La primera es **fuerza bruta**, la más básica, que es probar todas las posibles soluciones. Después tenemos **dividir y conquistar**, que trata sobre dividir el problema en subproblemas recursivamente, cada vez más chicos y fáciles, hasta que se llegue a un subproblema obvio. A partir de ahí la solución general se calcula combinando todas las subsoluciones. Un ejemplo de esto sería si queremos ordenar una lista; partimos la lista en dos, y así sucesivamente hasta que tengamos una lista de dos elementos, donde es obvio como se ordenan. Otra forma es la **programación dinámica** (no tiene nada que ver con los lenguajes dinámicos), que es parecida a dividir y conquistar, pero los

subproblemas se solapan y se deben recordar todas las subsoluciones para no recalcular. Existen otras técnicas que solo nombraremos, como **método codicioso**, la **programación lineal**, **reducción** a otro problema o **búsqueda y enumeración**.

Patrones de diseño

Anteriormente ya vimos algunos patrones de diseño, pero vimos pocos y todos de creación. Los patrones de diseño abarcan más el modo en cómo se construyen objetos, también muestran cómo deben colaborar para cumplir cierta tarea. Es indispensable que todo programador sepa al menos el nombre y el objeto cada uno de los patrones del **Gang Of Four**. Veamos algunos patrones adicionales que son muy conocidos y utilizados.

Empecemos con el patrón **Chain of Responsibility** (cadena de responsabilidades). Este patrón se basa en varios objetos polimórficos relacionados y forma una cadena cuyo propósito es tratar de resolver cierta petición. La idea es que cuando una petición o requerimiento llega a la cadena, el primero (o último, dependiendo como se haga) trata de resolverlo, y si no puede pasa el requerimiento al siguiente objeto en la cadena. Si algún eslabón en la cadena puede resolver la petición, lo hace y se devuelve el resultado. Si se llega al final de la cadena, que pasa cuando ningún objeto de ella puede resolver el pedido, se lanza un error o se indica de alguna otra forma que no se pudo resolver. También se puede poner un comportamiento por defecto al final de la cadena para que no dé error.

> SE PUEDE PONER UN COMPORTAMIENTO POR DEFECTO AL FINAL DE LA CADENA



PATRONES ESTRUCTURALES

Hemos olvidado de mencionar los patrones de diseño estructurales. Estos se centran en cómo se deben organizar las dependencias y la forma de estas en los objetos, con el fin de resolver una problemática determinada. Algunos de ellos son el **Composite** (Compuesto) que define una estructura polimórfica arbórea, y otro el **Adapter** (Adaptador), que es un **wrapper** para asociar objetos donde los protocolos no coinciden.

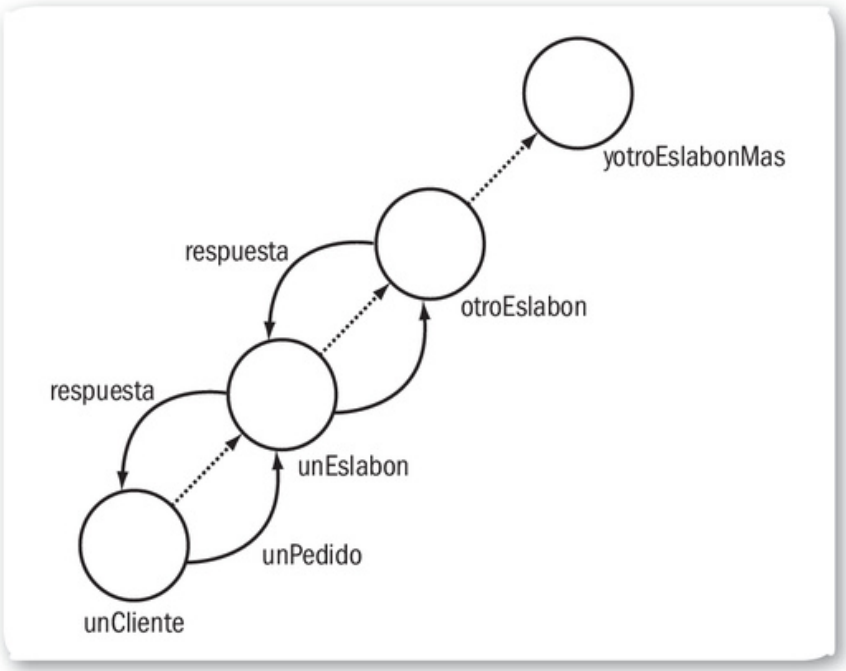


Figura 6. Así funciona un **`chain of responsibility`**, los pedidos se van pasando por los eslabones de la cadena hasta que uno lo resuelve.

Otro patrón interesante es el **Visitor** (Visitante). El **Visitor** trata de solventar la limitación de **single dispatch** (los métodos se resuelven dependiendo solamente del receptor) que tiene Java y muchos otros lenguajes orientados a objetos. Ahora, ¿qué pasa si queremos que un método se resuelva en base a dos o más objetos? Algunos lenguajes pueden hacer esto que se conoce como **multiple dispatch**. **Visitor** apunta al caso particular de dos objetos, que serían el receptor y el parámetro único del método. La idea detrás de la dinámica de las



MAVEN

Encontraremos a **Maven** en **http://maven.apache.org**. **Maven** es la herramienta de manejo de proyecto más famosa del mundo Java. Permite administrar las distintas fases de un proyecto, ya sea la construcción de los entregables, reportes, métricas de código, correr lo test unitarios y mucho más. Pero lo más importante es que permite manejar las dependencias sin problemas, ya que las descarga de sus servidores.

colaboraciones del patrón es que el receptor del mensaje delega en el parámetro enviándose a sí mismo como parámetro. Este nuevo envío de mensaje, donde ahora el receptor es el parámetro original, contiene la información (por ejemplo utilizando un nombre de método particular) del tipo del receptor original. De esta forma, en el segundo mensaje tengo los tipos de ambos objetos y, por lo tanto, se sabe qué método se tiene que ejecutar. Este patrón es muy utilizado, por ejemplo, para operar sobre estructuras arbóreas de objeto (no necesariamente polimórfico) y en la que se quiere operar de forma distinta con cada uno. Aunque no solo está limitado a ese tipo de uso, también se utiliza, por ejemplo, para resolver las operaciones aritméticas en algunos lenguajes. Por ejemplo, la suma de un entero y otro da un entero, pero un entero con un real da un real. Cada una es una operación distinta pero con un mismo nombre: suma. Este caso particular se conoce como **double dispatch**.

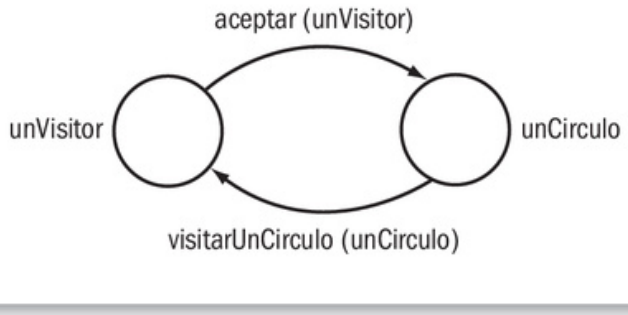


Figura 7. En el diagrama podemos apreciar las colaboraciones que definen al patrón **`Visitor`**. Se requieren dos mensajes para conocer ambos tipos.

Otro patrón particularmente útil es el **Strategy** o estrategia, en el cual se define que hay una serie de objetos polimórficos (jerarquía), llamados estrategia, que modelan cierto comportamiento, y hay otro objeto que colabora con uno de estos. La idea es que el objeto puede cambiar de estrategia en tiempo de ejecución, y simular un cambio de comportamiento. Un ejemplo es el ordenamiento, donde tenemos varios objetos que modelan cómo ordenar una lista de distinta formas, implementando distintos algoritmos. Entonces la lista utiliza una de estas estrategias según sea configurada o le convenga. Como la estrategia es

simplemente un objeto colaborador, puede ser dinámicamente cambiada. Otro patrón con una idea similar, la de aparentar un cambio de comportamiento en tiempo de ejecución, es el **Decorator** o decorador. Este patrón difiere del **Strategy** en que es un **wrapper** (envoltorio) del objeto. Tenemos básicamente dos objetos, el decorado y el decorador; la idea es que el decorador es polimórfico con el decorado, ya que cada mensaje que recibe es luego reenviado al decorado. De esta forma tenemos un objeto en el medio de los mensajes enviados al objeto que nos interesa. Ahora en este objeto decorador podemos hacer lo que queramos, y ampliar o modificar el comportamiento aparente del objeto decorado, ya que los clientes no tratan con este directamente, sino que lo hacen mediante el decorador (al ser polimórficos no notan la diferencia). Java utiliza este patrón en sus clases **streams** de **IO**. Lo bueno de este patrón es que, al ser polimórficos, puedo ir aplicando varias decoraciones sobre un mismo objeto.



Figura 8. En este diagrama podemos ver una representación que corresponde al patrón **Decorator**.

Existen muchos otros patrones y muchas fuentes, así que recomendamos al lector dedicarle un tiempo a este interesante tema.

VENTAJAS, DESVENTAJAS Y PROPÓSITO

Cuando estamos pensando en utilizar alguno de los patrones para encarar nuestro diseño, debemos pensar si realmente es el necesario. Hay varios patrones que son muy parecidos y atacan problemas similares. La diferencia entre ellos radica en su propósito, en las ventajas y desventajas. Debemos guiarnos primeramente por el propósito, para luego centrarnos en si nos conviene o no aplicarlo.

Threads y concurrencia

En los temas avanzados es inevitable hablar sobre la concurrencia. La concurrencia trata sobre cómo poder realizar varias tareas en simultáneo guardando la integridad de los recursos compartidos por las tareas. Desde el inicio de Java era posible ejecutar código en paralelo utilizando los métodos **wait**, **notify** y **notifyAll** declarados en **Object**, así como la interfaz **Runnable** y la clase **Thread**. Por desgracia, estas son herramientas muy básicas y hacían que el manejo de muchos **threads** de ejecución y su sincronización fuera realmente tedioso y muy propenso a errores muy difíciles de detectar. Afortunadamente en las recientes versiones de Java se agregó un paquete, el **java.util.concurrent**, que contiene una gran cantidad de clases e interfaces que proveen herramientas de alto nivel de abstracción para la sincronización y la ejecución en paralelo de código. Con estas nuevas herramientas, y el soporte anterior existente en Java, programar aplicaciones concurrentes ahora es mucho más sencillo.

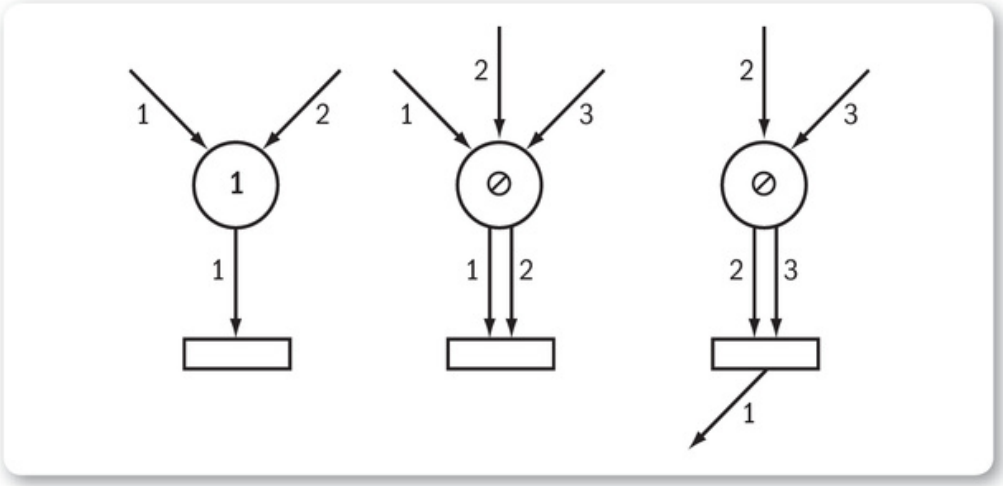


Figura 9. Un **semáforo** solo deja pasar cierto número de **threads**, mientras que los demás se quedan esperando hasta que alguno salga de la zona protegida.

Algunas de las clases que se proveen permiten manejar colecciones que son compartidas por varios threads de forma segura. Se definen las interfaces **BlockingQueue**, **BlockingDequeue**, **ConcurrentMap** y **ConcurrentNavegableMap** y sus respectivas implementaciones.

Además se ofrece una lista (**List**) y un conjunto (**Set**) que es seguro (**CopyOnWriteArrayList** y **CopyOnWriteArraySet**). Otras clases ofrecen una forma fácil de solicitar la ejecución de cierto código en paralelo y seguir ejecutando mientras se aguarda el resultado. Por ejemplo tenemos los **Executor**s que encapsulan el manejo de cuándo una tarea puede ser ejecutada en paralelo o no. También contamos con los **Future** (futuros) que representan el resultado de una ejecución en paralelo que puede, o no, haber concluido. Por último, encontraremos en este paquete clases para sincronizar en un punto del código varios threads. Primero están los semáforos representados por la clase **Semaphore**. Un semáforo tiene permitido dejar pasar un cierto número configurable de threads, cuando se supera este número, cualquier otro thread debe esperar hasta que alguno otro deje su lugar. Luego tenemos el **CountDownLatch**, que a diferencia de un semáforo, que bloquea cuando se supera un número de threads, este funciona al revés. Un **CountDownLatch** necesita un cierto número de activaciones (enviándole el mensaje **countDown**) hasta que se libera y todos los threads que estaban esperando empiezan a ejecutar.

> FUTURE REPRESENTA EL RESULTADO DE UNA EJECUCIÓN EN PARALELO

La mayor cantidad de errores que se producen cuando estamos en presencia de varios threads es cuando tenemos un estado compartido. Cuando varios threads pueden consultar y, específicamente, modificar un mismo objeto, es muy probable que se rompan los invariantes del objeto y de los threads. Supongamos que tenemos un objeto persona que tiene una cierta edad, por ejemplo, catorce años, y un thread consulta este valor y decide que esa persona tiene que cumplir años. Ahora otro thread hace lo mismo antes de que el primero haga efectivamente que



PROGRAMACIÓN CONCURRENTE

Cuando estamos pensando una aplicación y creemos que tenemos que hacer **multithread**, hay que pensar una y otra vez si es necesario. Existen varias alternativas, librerías, frameworks y tecnologías que nos pueden evitar tener que lidiar nosotros mismos con este tema tan complicado. Un camino inicial que nos puede ayudar a paralelizar en el futuro es utilizar ampliamente la inmutabilidad.

se aumente la edad de la persona. Resulta que ambos threads creen que la persona tiene catorce años y le piden que cumpla años dos veces. En consecuencia la persona termina teniendo dieciséis años en vez de quince. Una forma de evitar este tipo de problemas es haciendo que los threads no compartan información o, si lo hacen, esta sea inmutable. Con objetos inmutables estos problemas directamente no existen.

Localización e internacionalización

En la actualidad, cualquier aplicación que se precie, ya sea de escritorio o web, tiene que soportar distintos idiomas. No solamente importa el idioma del texto, sino que también es importante para el usuario ver las fechas, los números, los montos y otras medidas en la forma habitual para él, en la norma que se utiliza en su país. Esto se conoce como **internacionalización** y **localización**. El primer término significa construir un sistema que pueda ser adaptado a distintos lenguajes y regiones sin incurrir en cambios, el segundo, se refiere al hecho de escribir los componentes propios de cada lenguaje y también la región para un sistema que sea internacionalizado.

> EN LA ACTUALIDAD CUALQUIER SISTEMA DEBE SOPORTAR DISTINTOS IDIOMAS

Java soporta internacionalización inherentemente, ya que clases como **Date** están afectadas por la localización del sistema que está ejecutando la maquina virtual. La clase que se encarga de esto es **Locale**, y representa una región geográfica, política o cultural específica. Se utiliza en las operaciones que sean sensibles de localización. Por defecto se toma la configuración de la máquina en la que se está ejecutando pero es



UTILIZAR UN FRAMEWORK

Cuando queremos trabajar en una aplicación internacionalizable debemos tener en cuenta el uso de un framework que nos facilite la tarea de administrar los textos en los distintos idiomas y nos dé alguna herramienta que simplifique el formateo. **Spring** es uno de esos frameworks que nos dan una gran ayuda, ya que se encarga él mismo de obtener la configuración correcta, los textos y de formatear.

posible modificar esto en tiempo de ejecución o utilizar otro **Locale** a voluntad. En general, cuando internacionalizamos, lo que se busca es mostrar las fechas y números en el formato adecuado a la región y los textos en el idioma correspondiente. Para lo primero, el formateo, se utiliza la familia de clases de **Format**. Esta jerarquía define el protocolo básico para obtener la representación textual localizada de objetos (formatear) y luego, en base a la representación textual, recuperar el objeto (parsear). Las subclases de **Format** se especializan en distintos tipos de objetos. **DateFormat** (y más específicamente su subclase **SimpleDateFormat**) se encarga del formateo de las fechas.

```
// lo queremos en español
final Locale locale = new Locale("es");

// especificamos un patrón para la fecha
final String patron = "EEE, MMM d, 'yy";

// obtenemos el formateador
final DateFormat formateador = new SimpleDateFormat(patron, locale);

// vamos a formatear el primero de mayo del 2011
final Date fecha = primeroDeMayoDel2011();

// formateamos
final String texto = formateador.format(fecha);

// verificamos
assertEquals("dom, may 1, '11", texto);
```

Es importante señalar que otra clase dedicada al formateo es **MessageFormat** que se encarga de darles formato a mensajes que contienen distintos tipos de objetos. Por ejemplo, supongamos que queremos armar el texto "Hoy es 15 de abril del año 1920, la población mundial de peces payaso es de 1.000.000 de especímenes". Tenemos por un lado el objeto que representa la fecha y por otro la cantidad de peces. Una opción es armar el texto a mano formateando primero la fecha y luego la cantidad para, finalmente, componer el texto a mano.

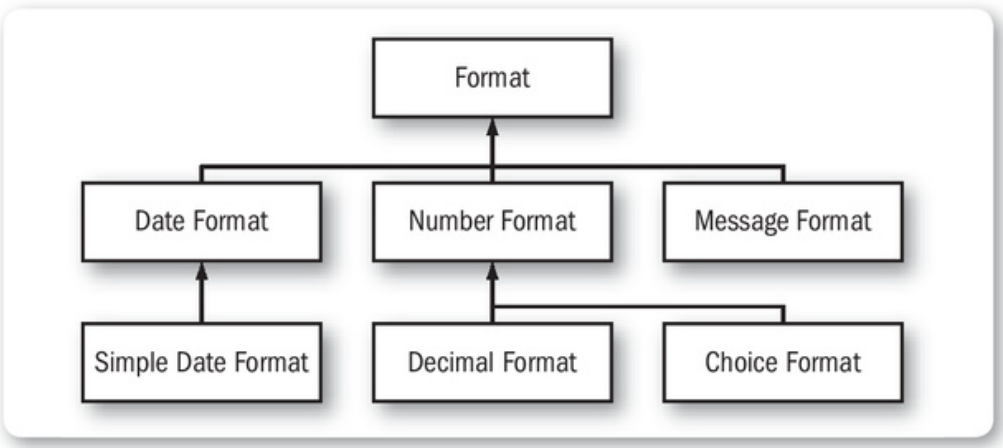


Figura 10. Esta imagen nos presenta un diagrama de clases que corresponde a la jerarquía de **Format**.

MessageFormat simplifica esta tarea, solo necesitamos pasarle el patrón del texto con indicadores de dónde queremos poner la fecha y el número, los ingresamos y él los formatea y compone automáticamente.

```
// generamos el patrón
String patron = "Hoy es {0,date,d 'de' MMMM 'del año' yyyy}, la población mundial
de peces payaso es de {1,number,integer}{2,choice,1# especimen|1< especimenes}";

// elegimos la fecha
Date fecha = quinceDeAbrilDe1920();

// y la cantidad
long cantidad = 1000000L;

// instanciamos el formateador
MessageFormat formateador = new MessageFormat(patron, new Locale("es"));

// y formateamos
String texto = formateador.format(new Object[] {fecha, cantidad, cantidad});
```

MessageFormat utiliza las llaves para delimitar dónde van los objetos que se van a formatear; estos huecos o **placeholders** llevan el índice

del objeto cuando lo pasamos en el array por formatear. También se le puede indicar con qué formateador queremos hacer este procedimiento. En este caso indicamos **date** para usar el **DateFormat**, **number** para indicar **NumberFormat** y, finalmente, **choice** para un **ChoiceFormat**.

La clase **NumberFormat** se encarga de definir el protocolo de formateo y parseo de números, tiene dos subclases. La primera, **DecimalFormat** es la clase principal para tratar con los números directamente. Esta clase no hace uso directo del **locale** sino que lo hace a través de otro objeto de la clase **DecimalFormatSymbols**, que define los símbolos (separador de miles, de decimales, etcétera) que se usarán para formatear.

```
// instanciamos los símbolos
DecimalFormatSymbols simbolos = new DecimalFormatSymbols(new Locale("es"));

// instanciamos el formateador
DecimalFormat formateador = new DecimalFormat("#,###.##", simbolos);

// y formateamos
String texto = formateador.format(1000000L);
```

La clase **ChoiceFormat** se utiliza principalmente para cuando queremos pluralizar un texto en base a una cierta cantidad. En el ejemplo utilizamos la forma de patrón para indicar cuándo usar singular y cuándo plural. También lo podemos hacer programáticamente.

```
// instanciamos el formateador
ChoiceFormat formateador = new ChoiceFormat(
    new double[] {1, ChoiceFormat.nextDouble(1)},
    new String [] {"especimen", "especimenes"});

// verificamos el plural
assertEquals("especimenes", formateador.format(1000000L));

// y el singular
assertEquals("especimen", formateador.format(1L));
```

Para completar la internacionalización de la aplicación, debemos mover los textos visibles para el usuario a los archivos de recursos. Estos se conocen como **Resource Bundles** y se utilizan como distintos archivos de propiedades (donde cada línea está compuesta por **key=value**) con un mismo nombre principal pero con un sufijo locale agregado, de tal forma que cuando pedimos el recurso, especificando el **locale**, obtenemos los textos correspondientes. Luego en nuestro código solamente hacemos uso de los nombre de los textos (**key**).

```
// archivo red/user/java/textos_es.properties
saludo=Hola!!

// archivo red/user/java/textos_en.properties
saludo=Hello!!

// levantamos los textos en español
ResourceBundle español = PropertyResourceBundle.getBundle(
    "red.user.java.textos", new Locale("es"));

// y los textos en inglés
ResourceBundle ingles = PropertyResourceBundle.getBundle(
    "red.user.java.textos", new Locale("en"));

// verificamos el texto en español
assertEquals("Hola!!", español.getString("saludo"));

// y luego el texto en inglés
assertEquals("Hello!!", ingles.getString("saludo"));
```



PENSAR UNA APLICACIÓN LOCALIZADA

Cuando creamos una aplicación, por más que no tengamos pesado hacerla para muchos idiomas, igualmente deberíamos externalizar todos los textos en archivos de recurso. De esta forma, en un futuro, si queremos o necesitamos localizar la aplicación, ya tendremos la base y solamente deberíamos traducir los textos y modificar una pequeña porción de código. Prever estos recursos no requiere trabajo extra y trae beneficios.

Desarrollo de aplicaciones para la consola de comandos

Las aplicaciones de consola son escasas en esta época, pero todavía son habituales en la programación y administración de servidores. Con Java es posible hacer aplicaciones que interactúan con la consola, tanto para recepción de comandos como para la escritura en pantalla. En su forma básica, se define en una clase un método estático llamado **main** que recibe los argumentos con los que ejecutamos el comando.

```
public class MiComando {
    ...
    public void main(String [] argumentos) {
        if(args.length == 0) {
            System.err.println("Pasar un argumento");
            return;
        }
        final String nombre = args[0];
        System.out
                .append("Hola ")
                .append(nombre).println();
    }
    ...
}
```

Hay que tener en cuenta que para ejecutar nuestro programa debemos acceder a la consola de comandos y ejecutar.



JAVACURSES

Esta librería alojada en **http://sourceforge.net/projects/javacurses** está basada en la librería **ncurses** existente para **UNIX** que permite crear aplicaciones de consola que tengan interfaz gráfica centrada en caracteres. Permite crear menús, ventanas y botones totalmente renderizados con caracteres en la consola. Esta librería es muy útil para desarrollar aplicaciones de administración para servidores.

```
java mi.paquete.MiComando argumento1 argumento2 ...
```



Figura 11. En esta imagen vemos la ejecución de nuestro comando de consola, sin argumentos y con un argumento.

En Eclipse podemos también ejecutar este comando, esta aplicación de consola, desde el menú de **Run** o **Debug**.

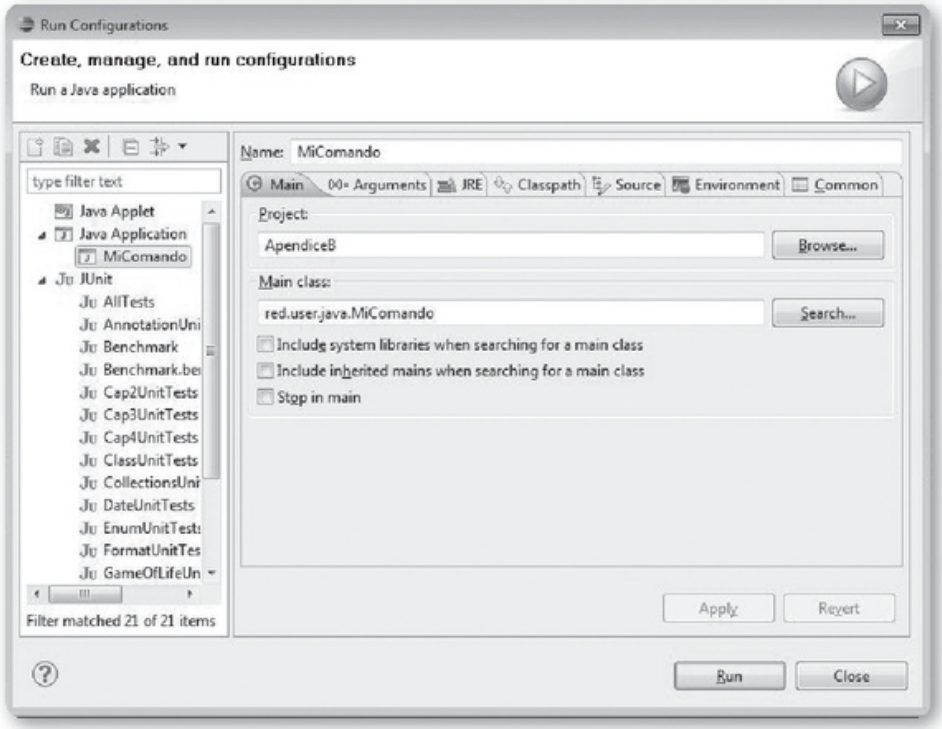


Figura 12. Aquí podemos ver la ventana de lanzamiento de aplicaciones que nos muestra **Eclipse**.

En nuestro código los argumentos se obtienen del array de **String**s que recibe el método **main** en el orden que fueron pasados al comando. Para interactuar con la consola, Java pone a nuestra disposición tres **streams**; uno es el de lectura, que permite leer los caracteres que se ingresan en ella. Este stream se accede en **System.in**, es un atributo público estático en la clase **System**. Luego, para escribir en la consola hay dos **streams** disponibles, el **System.out** para escritura normal y el **System.err** para escritura de errores (la consola puede discriminar entre estos dos).

Debemos saber que para manejos más avanzados de la consola existen librerías que nos ayudan en esta tarea.

Desarrollo de aplicaciones de escritorio

Si bien las aplicaciones de escritorio hechas en Java son escasas, es perfectamente factible su desarrollo. Java ofrece varias alternativas para la creación de ventanas, botones y otros elementos pertenecientes a la interfaz de usuario que encontramos habitualmente en las aplicaciones de escritorio.

El Abstract Window Toolkit

El **AWT** fue el primer sistema que incluyó Java para la construcción de aplicaciones de escritorio **multiplataformas**. Ofrecía las funcionalidades básicas requeridas por una aplicación; tenía botones, cajas de texto, menús y todo tipo de controles. Este **toolkit** abstraía al



SWT

En el sitio **www.eclipse.org/swt** podremos obtener todo lo que necesitamos para empezar a desarrollar nuestras aplicaciones utilizando este fantástico **toolkit**. Podremos descargar la librería para Java y las versiones nativas para las distintas plataformas como **Windows**, **Linux** y **Mac OS X**. También encontraremos tutoriales sobre los distintos **widgets** y enlaces a varias otras herramientas.

programador de los distintos modelos gráficos que tiene cada sistema operativo. Sin embargo cada aplicación se veía y funcionaba como si fuera nativa. Esto es más una restricción, ya que se ofrecían algunos controles básicos y se dejaban de lado controles más avanzados.



Figura 13. Esta imagen nos presenta un ejemplo de interfaz gráfica lograda con el uso de **AWT**.

Swing

A pesar de tener el **AWT**, más tarde Java introdujo este nuevo **toolkit** para el desarrollo de aplicaciones gráficas. A diferencia de **AWT,** que era parte en Java y parte nativo al sistema operativo, **Swing** está íntegramente hecho en Java. Además de todos los controles ya existentes en **AWT**, **Swing** introdujo muchos otros controles modernos, como paneles con tabs, que también permitían modificar la vista de los controles y las aplicaciones usando temas y pieles que lograban darle una apariencia distinta a todas las plataformas.



LOG4J

En el sitio web **http://logging.apache.org/log4j** encontraremos la librería de **logging** más popular del mundo. **Log4j** nos permite fácilmente llevar cuenta de lo que va pasando en nuestro código (de los errores principalmente). Todo lo que vamos escribiendo mediante configuración puede ser dirigido a distintos lugares, ya sea un archivo, un e-mail o una base de datos. Muy útil para saber si algo salió mal.

USERS PRESENTA...

¡EL PRIMER EBOOK USERS!

Sí, ya podés leer Hackers al descubierto en tu PC, notebook, Amazon Kindle, iPad, en el celular...

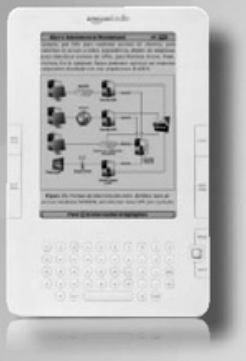
CONSEGUILO DESDE CUALQUIER PARTE DEL MUNDO

A UN PRECIO INCREÍBLE



¿QUÉ ESTÁS ESPERANDO?







¡LEELO DONDE QUIERAS!

INGRESA YA A USERSHOP.REDUSERS.COM Y ENTERATE MÁS

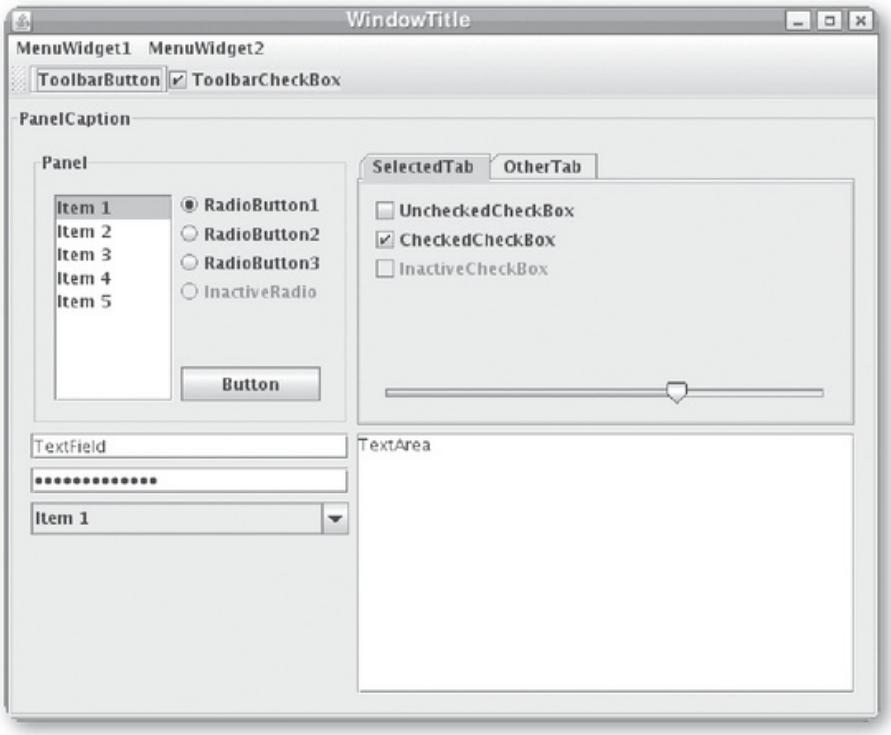


Figura 14. En esta imagen podemos ver un ejemplo de una interfaz gráfica obtenida con el uso de **Swing**.

El Standard Widget Toolkit

El **SWT** es otro **toolkit** para desarrollar interfaces gráficas en Java bastante popular, ya que es el utilizado por **Eclipse**. Creado con el propósito de sacar el mayor provecho de las plataformas nativas, tanto en los controles como en la velocidad. Originalmente pensada para solucionar las limitaciones del **AWT** en cuanto a controles y los problemas de velocidad que presentaba **Swing**.

JAVA MICRO EDITION

Para encontrar esta distribución de Java para dispositivos móviles debemos dirigirnos en nuestro navegador a **www.oracle.com/technetwork/java/javame/index.html**. Allí encontraremos el SDK con el entorno de desarrollo y el emulador de celular. Además el sitio cuenta con una amplia biblioteca de tutoriales, artículos y documentación sobre todas las características de esta versión reducida de Java.

Es necesario que tengamos en cuenta que los programas escritos con este **toolkit** son portables mientras la librería esté disponible para la plataforma en la cual deseemos ejecutar las apliccaiones.



Figura 15. Ejemplos de interfaces gráfica con **SWT** funcionando en distintas plataformas **Vista**, **XP** y **Linux**.

Desarrollando aplicaciones para dispositivos móviles

En el último tiempo hemos presenciado un boom en las tecnologías móviles de la mano de los **smartphones** y de las **tablet pc**. Por eso la demanda de desarrollos móviles es cada vez mayor.

Java Micro Edition

Java ofrece desde hace muchos años una versión reducida de la API normal que apunta a celulares y **PDA**s. Esta versión se conoce como **Micro Edition** y permite crear desde interfaces de usuario hasta enviar **mensajes de texto** o interactuar con otras capacidades de los dispositivos, como por ejemplo el uso del **Bluetooth**. Esta distribución de Java fue muy utilizada en los celulares antiguos. En la actualidad todavía goza de un gran soporte en varias plataformas.

De la página de **Oracle** es posible bajar esta distribución que viene ya con un entorno de desarrollo y también de prueba.



Figura 16. En esta imagen podemos ver una captura de pantalla del simulador de celular.

Android

Actualmente está teniendo mucha repercusión esta nueva plataforma creada por **Google** como el sistema operativo para cualquier dispositivo, y que compite con el **iOS**, el sistema operativo del **iPhone**. A diferencia de la edición **Micro** de Java, **Android** es un sistema con el que el usuario interactúa y las aplicaciones son programadas en Java utilizando una API especial para acceder a todas la funciones del dispositivo móvil, ya sea la **cámara**, el **GPS** o el **acelerómetro**. La mayoría de los **smartphones** nuevos utilizan **Android** como plataforma.



ANDROID

En el sitio web que encontramos en la dirección **www.android.com** se pone a disposición de los usuarios toda la información respecto de esta plataforma de **Google** para dispositivos móviles. Tendremos acceso a las distintas plataformas soportadas, a todo lo necesario para el desarrollo, como las librerías, el entorno y tutoriales; y finalmente al **Android Market**. El **Android Market** es el lugar donde los usuarios pueden descargar las aplicaciones pagas y gratuitas para **Android**.



Figura 17. Logo del sistema operativo **Android**.

Desarrollando aplicaciones web

La Web, indiscutiblemente, es el centro del universo para los programadores en la actualidad. Las aplicaciones web han ido incrementado su complejidad a lo largo de estos años, cada vez con más funcionalidades que ya compiten con las aplicaciones de escritorio. Además tienen que soportar miles o millones de usuarios simultáneamente, con comunicación en tiempo real entre ellos y el sistema. Java debe su fama actual y su larga vida a que enfocó sus esfuerzos en promoverse como una tecnología, no solo para la Web y para los servidores, sino que específicamente para aplicaciones empresariales.

> CON EL TIEMPO LAS APLICACIONES WEB HAN IDO AUMENTANDO SU COMPLEJIDAD

Los servlets

Los **servlet** son la pieza básica de código que se puede escribir en Java para programar para la Web. Antiguamente la Web era más bien estática o con algo de dinamismo gracias a algunos scripts de **C** o de **Perl** que se ejecutaban en el servidor y cuya salida era directamente enviada al navegador del cliente. Estos scripts no solo tenían que proveer cierta funcionalidad, sino que también tenían que generar el **HTML** que formaba la página que veía el cliente. Los **servlets** son la versión Java de esos scripts y son la primera línea de abstracción que

tiene el programador, al tener acceso al pedido del browser y al **stream** de salida para enviar la respuesta. Al igual que con los antiguos scripts, los **servlets** tienen que cumplir con su función y, además, generar la respuesta. Claramente son dos responsabilidades distintas y, por lo tanto, tener ambas cosas en un solo objeto hace que este sea difícil de programar y mantener. Además el **HTML** de respuesta tiene que ser armado concatenando **String**s. Este problema de armar la página, fue luego aplacado con la aparición de las **Java Server Pages** o **JSP**. Las **JSP** son un **template**, una plantilla, donde se define la página web utilizando una mezcla de **HTML** y **tags** propios, más algo de código Java. Los **JSP** son luego compilados a **servlets**. De esta forma se separa entonces la lógica de comportamiento, de la lógica de visualización de la página.

LOS SERVLETS REQUIEREN DE UN SERVIDOR O CONTENEDOR DE SERVLETS

Los **servlets** requieren de un servidor Java llamado **contenedor de servlets**, los cuales son servidores web completos, hechos en Java, y que permiten definirle distintos **servlets**.

Existen muchos **frameworks** que sobre estas tecnologías ofrecen muchas más facilidades y abstracciones para desarrollar aplicaciones web. Algunas ofrecen **templates** más avanzados, con más funcionalidades. Otros abstraen aún más el nivel y definen objetos, llamados **widgets**, que además establecen cómo se tienen que ver, y el programador se dedica a componer **widgets** en la página. Es necesario que tengamos en cuenta que esta forma de proceder hace que la programación de aplicaciones Web sea similar al desarrollo de una aplicación de escritorio, de esta forma podremos enfrentarnos a ella de una forma más sencilla.



TOMCAT

Tomcat es el contenedor de **servlets** más conocido y utilizado que hay. Ya desde hace años es el contenedor preferido por la comunidad Java. Lo podemos encontrar en la dirección web **http://tomcat.apache.org** donde no solo encontraremos el servidor, sino también su código fuente y mucha documentación al respecto. Es muy fácil instalarlo y no requiere configuración para empezar a funcionar, pero es altamente configurable para escenarios más serios y profesionales.

Los Enterprise Java Beans versión 3 (EJB3)

Lo **EJB** (versiones 1 y 2) fueron una tecnología que Java ofrecía para el desarrollo de aplicaciones empresariales que apuntaban a resolver el tema de la distribución del sistema en varias ubicaciones distintas y cómo mantener los objetos en una base de datos, entre otras cosas. Lamentablemente fue una implementación deficiente y prontamente (aunque el daño ya estaba hecho sobre miles y miles de aplicaciones creadas) obtuvo la fama de ser innecesariamente complicado y muy lento. Por suerte esto dio pie a la popularización de otras alternativas, lo que originó el framework **Spring**.

> EJB3 ESTANDARIZÓ LAS DISTINTAS TECNOLOGÍAS PRNCIPALES EN CADA ÁMBITO

Luego de varios años, apareció la versión 3 de los **EJB**. Esta versión, en vez de proponer una nueva forma de resolver los problemas, estandarizó las distintas tecnologías principales en cada ámbito. Utilizando como herramienta principal las anotaciones, el estándar **EJB3** define cómo inyectar las dependencias, como se mantienen los objetos en la base de datos y varias otras cosas, como el alcance de un objeto o su exposición como **web service**. En realidad todas estas anotaciones y su propósito están basados en las definidas por otros frameworks como **Spring** y **Hibernate**, al ser estos la implementación por defecto. Lo bueno de esta unificación de tecnologías bajo un mismo estándar logra que las herramientas, ya sea servidores, librerías o IDE, puedan proceder a realizar un trabajo utilizando los mismos conceptos y no tengan que duplicar esfuerzos para soportar distintas tecnologías. Así todo tiende a realizarse de una forma mucho más sencilla.



JBOSS

JBoss, **www.jboss.org**, es una comunidad **Open Source** que se dedica al desarrollo de la plataforma Java en el ámbito empresarial. Su mayor contribución es el **Application Server** Java del mismo nombre. Este servidor no solo es un contenedor de **servlets** sino que también implementa el estándar **EJB3**, con lo cual podemos desarrollar aplicaciones empresariales con esta tecnología.

El Google Application Engine

El **Google Application Engine** o **GAE** es la infraestructura de múltiples servidores de **Google** para **cloud computing**. Google nos permite subir nuestra aplicación web gratuitamente (con planes pagos también) a sus servidores de alta disponibilidad. El **GAE** ofrece un entorno Java de servidor bastante completo (con algunas limitaciones) y con acceso a algunas formas de persistencia basadas en las tecnologías de **Google**. La ventaja de este alojamiento es que nuestra aplicación puede escalar utilizando la misma plataforma que otras aplicaciones del gigante de las búsquedas. Si estamos pensando en nuestra primera aplicación web, esta es una gran opción para tener en cuenta. Hay que estudiar sí las limitaciones de tiempo de ejecución y memoria que se imponen, además de ver que algunas clases del Java SDK prohibidas no creen un conflicto con el objetivo de nuestra aplicación. Las tecnologías **NoSQL** (no hay base de dato relacional) que ofrece el **GAE** incluyen el **Big Table** y **Blob Storage**.

GAE OFRECE UN ENTORNO JAVA DE SERVIDOR COMPLETO PERO CON ALGUNAS LIMITACIONES



Figura 18. Logo del **Google Application Engine**.

GOOGLE APPLICATION ENGINE

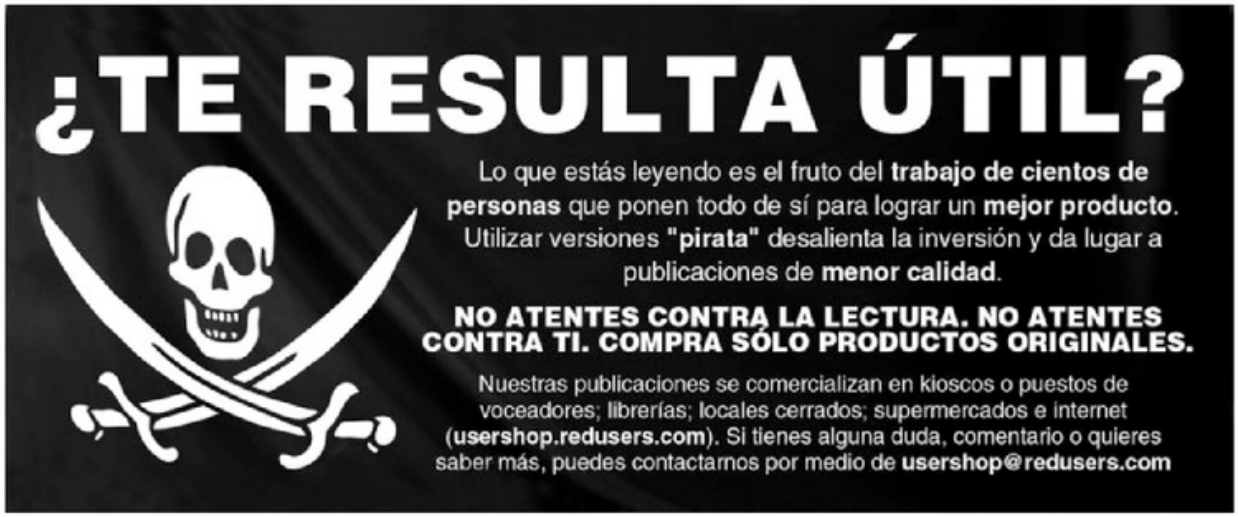
Localizado en la dirección web **http://code.google.com/appengine**, encontraremos todo lo necesario para empezar a programar nuestra aplicación web. Allí están las librerías básicas de **Google**, un **plugin** para **Eclipse**, mucha documentación sobre todas las **API** que provee el **GAE**. También hallaremos una gran cantidad de tutoriales, tanto escritos como videos sobre diversos temas, desde cómo desarrollar nuestra primera aplicación hasta cómo subirla a la nube.

Otros lenguajes para la JVM

Ahora que ya hemos aprendido a desarrollar en Java, podemos investigar algunos otros lenguajes que corren sobre una máquina virtual. Estos lenguajes ofrecen mayores facilidades a la hora de programar aunque generalmente esto provoca algunos inconvenientes en la performance. Es interesante estudiar otros lenguajes para aprender sobre las fuerzas y debilidades de cada uno, enriquecernos como programadores, ya que así tendremos a nuestro alcance un mayor arsenal para enfrentarnos a nuevos desarrollos.

Groovy

Este lenguaje es bastante popular debido al framework de desarrollo web veloz llamado **Grails**. **Groovy** es un lenguaje orientado a objetos que se encuentra por encima de Java, esto quiere decir que todo lo que es válido en Java, también lo es para **Groovy**, por esta razón puede ser una muy buena alternativa si queremos profundizar en otro lenguaje. La particularidad de **Groovy** es que es un lenguaje dinámico, que no requiere que especifiquemos los tipos de las referencias. Además, tiene un sistema de reflexión y **metaprogramación** muy poderoso ya que podemos modificar en tiempo de ejecución casi cualquier cosa, desde los métodos de un objeto hasta su clase. Debemos tener en cuenta que también ofrece una sintaxis más laxa que la de Java, con soporte para lista, mapa y clausuras de forma nativa.



Scala

Se trata de una opción considerada por muchos como el Java de la siguiente generación, **Scala** es un lenguaje muy distinto de Java, que tiene nuevas abstracción y también formas de tipado, como los denominados **traits**. **Scala** tiene un poderoso y moderno compilador que hace que no sea necesario especificar tipos (ya que él los deduce automáticamente) y que podamos escribir nuestros programas de una forma muy simple. **Scala** hace un gran hincapié en la programación de objetos, ya que aquí realmente todo es un objeto; por ejemplo no existen los métodos estáticos y las clases son objetos. Así mismo hace una fuerte apuesta por la programación funcional (otro paradigma de la programación), de forma que las funciones tienen el mismo peso en el lenguaje que otros elementos, como los métodos, y estas son realmente objetos también (podemos mandarles mensajes). Del ámbito funcional también tiene el denominado **pattern matching**, donde es posible elegir distintas ejecuciones de acuerdo a la forma que definimos para el objeto. debemos tener en cuenta que **Scala** ofrece muchas ventajas respecto de Java y lo hace sin perder casi nada de performance.

SCALA ES UN LENGUAJE MUY DISTINTO DE JAVA, QUE OFRECE MUCHAS VENTAJAS

```
// multiplica los números del 1 al 9, da 362880
1 to 9 foldLeft 1 (_ * _)
```

RESUMEN

Estos son algunos de los temas que el lector puede seguir investigando y estudiando para poder crecer como programador. No solo como programador enfocado en Java, sino que como programador a un nivel más abstracto, independiente del lenguaje en que se programe. Java es un lenguaje que abarca varios ámbitos, desde aplicaciones de consola hasta celulares, pasando por el escritorio y el servidor. Así que es cuestión del lector decidir su camino y por dónde va seguir a continuación.

Servicios al lector

En esta sección nos encargaremos de presentar un útil índice temático para que podamos encontrar en forma sencilla los términos que necesitamos.

▾ Índice temático..... 310

Indice temático

#

@deprecated 203
@Deprecated 203
@Inherited 208
@Memoize 216
@Override 85, 128
@Retention 206
@Target 206, 207
@Test 40, 42, 202, 212

A

Abierto/Cerrado 121
Abstract 46, 77, 95, 264
AbstractCollection 252, 258
AbstractList 252
Abstraer 20
Alan Kay 14
Algol 60, 14
Alternativas a la herencia múltiple 25
Annotation 210
Anonymous classes 102
API 156, 246
Append 176, 188
Apple Lisa 14
Applet 277
Área de trabajo 34
Argumentos 17
Array 83, 140, 179, 192
ArrayList 252
ArrayQueue 281
ASP 31

B

Beans 200, 226
Big Table 306
Bluetooth 301
Boolean 46, 51, 68, 80, 108, 174
Break 46, 47, 50, 54, 64

C

C 15, 303
C++ 16, 21, 24, 30. 48
Calendar 186, 188
Capacidades reflectivas 14
Catch 46, 51, 65, 152, 154, 155
Cell 107, 109, 119
Char 47, 68, 176, 204
Choice 294
ChoiceFormat 294
Ciclo de modelado 26
Class 40, 47, 76, 77, 126, 136
ClassLoader 274, 275, 277, 278
Collection 179, 180, 181
Comparable 140, 281
compareTo 257

D

Date 186, 188, 245, 257, 258, 291
DeadCell 107
Debug 114
Decimal 68
Dequeue 281
Despacho dinámico 19
Dividir y conquistar 284
Double 47, 59, 69, 204
Do while 46, 51, 52
Duke 31
Dynamic dispatch 19

E

Eclipse 300
Eclipse ID 33
EJB 200, 305
EJB3 305
ElementType 206
Else 47, 61, 62
Empleado 80

E

Encapsula 16
Entrerprise Java Beans 200
Entry 182
Enum 139, 140
Equals 71, 172, 173, 174, 180
Exception 155, 157
Executor 290
Extends 47, 76, 128, 167, 168

F

False 46, 48, 53, 173
File 38, 126, 207
Final 47, 59, 77, 82, 102, 264
Finish 38, 126, 136, 207
First In, First Out 280
Flash 30
Float 69, 175
For 46, 48, 52, 53, 54, 56, 57
For each 52, 57, 58, 178, 181, 185
forName 254
Frontera 18

G

GameOfLife 106, 118, 120
Garbage collector 30
Get 79, 80, 181, 268
Google 31
GPS 302
Green 30

H

Handler 151, 152, 154, 155
hashCode 172, 173, 180
Herencia entre clases 21
Herencia múltiple 21

I

IEEE 754 47, 50, 69
if 47, 48, 61, 62
IllegalArgumentException 140
Implements 48, 76
Import 48, 49, 72, 73

I

Inner classes 98, 101
InputStream 192, 193, 195
Int 48, 69, 71, 204
Integer 58, 69, 175
Interface 48, 204, 265
intValue 58
InvocationHandler 274, 276
Invoke 259, 268, 274
iPhone 302
Is 79, 80, 108
Isomorfismo 27

J

Java 16, 151
JavaBeans 200
java.util.concurrent 289
Jerarquía mal utilizada 23
Joda Time 187
JVM 49, 58

L

Last In, First Out 280
List 179, 280, 290
Lista 22
Long 48, 69, 71, 204

M

Map 181, 280
Mappable 183
Máquina virtual 30
Mensaje 18
MessageFormat 292, 293
Metadata 43, 200, 210
Method 256, 274
Método 17
Modifier 264

N

name() 139
Native 48, 82
Nested class 99
New 48, 86

N

Next 58
NoSQL 306
NoSuchMethodException 255
Null 52, 173, 179, 181, 268
Number 174

O

Object 47, 58, 76, 96
Object Relational Mapping 222
Of 145
Outline 35

P

Package 49, 72
Package Explorer 38
Paquetes 72
Paradigma de objetos 26
Pattern matching 308
Performance 116
Pila 22
PM 190
PriorityQueue 281
Protected 49, 81, 107
Proxy 273, 276
Pulsar 103

R

Receptores 18
Reducir uso directo de clases 114
Refactoring 14
Row 107
Run 297
RuntimeException 153, 157, 158

S

Servlets 31, 303, 304
Set 144, 180, 181, 280, 290
setAccessible 263
Setter 79, 219
Short 49, 69, 71, 204
Singleton 241, 242, 243
Smalltalk 15

S

Spring 227, 305
Static 48, 49, 73, 81, 82, 97, 268
Static factory methods 105
Stream 192, 193, 194
String 69, 70, 134, 174, 187, 218, 252
Subtipos 19
Superclase 21
Supertipo 20
Swing 299, 300
Switch 46, 47, 50, 64, 143
Synchronized 50, 82, 264

T

TDD 36, 188
testAdd 40, 42
Test unitarios 14
This 50, 81, 87, 90, 256, 268
Threads 50, 289
Throws 51, 82, 157, 204
Timeout 202, 212
Tipos 17
Toolkit 298, 300
toString 114, 172
True 46, 48, 52, 5
Try 46, 47, 51, 154, 155

U

Unicode 47, 68, 176
UnitTests 38
UnsupportedOperationException 179

V

Validar creación 112
Value 164, 205, 206
Virtual machine 33
Void 40, 43, 51, 82, 85, 202, 259
Volatile 51

W

Web service 305
While 46, 47, 51, 52, 54, 55
Wizard de Eclipse 76

CLAVES PARA COMPRAR
UN LIBRO DE COMPUTACIÓN

1 SOBRE EL AUTOR Y LA EDITORIAL

Revise que haya un cuadro "sobre el autor", en el que se informe sobre su experiencia en el tema. En cuanto a la editorial, es conveniente que sea especializada en computación.

2 PRESTE ATENCIÓN AL DISEÑO

Compruebe que el libro tenga guías visuales, explicaciones paso a paso, recuadros con información adicional y gran cantidad de pantallas. Su lectura será más ágil y atractiva que la de un libro de puro texto.

3 COMPARE PRECIOS

Suele haber grandes diferencias de precio entre libros del mismo tema; si no tiene el valor en tapa, pregunte y compare.

4 ¿TIENE VALORES AGREGADOS?

Desde un sitio exclusivo en la Red, un Servicio de Atención al Lector, la posibilidad de leer el sumario en la Web para evaluar con tranquilidad la compra, y hasta la presencia de adecuados índices temáticos, todo suma al valor de un buen libro.

5 VERIFIQUE EL IDIOMA

No solo el del texto; también revise que las pantallas incluidas en el libro estén en el mismo idioma del programa que usted utiliza.

usershop.redusers.com
VISITE NUESTRO SITIO WEB

» Vea información más detallada sobre cada libro de este catálogo.
» Obtenga un capítulo gratuito para evaluar la posible compra de un ejemplar.
» Conozca qué opinaron otros lectores.
» Compre los libros sin moverse de su casa y con importantes descuentos.
» Publique su comentario sobre el libro que leyó.
» Manténgase informado acerca de las últimas novedades y los próximos lanzamientos.

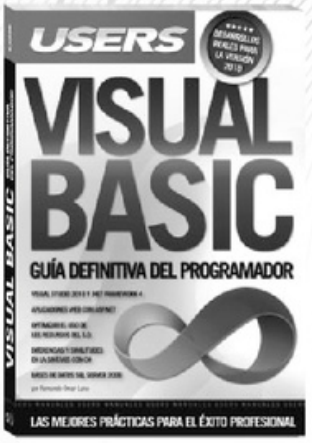
TAMBIÉN PUEDE CONSEGUIR NUESTROS LIBROS EN KIOSCOS O PUESTOS DE PERIÓDICOS, LIBRERÍAS, CADENAS COMERCIALES, SUPERMERCADOS Y CASAS DE COMPUTACIÓN.



LLEGAMOS A TODO EL MUNDO VÍA »OCA * Y DHL **
* SOLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

usershop.redusers.com // usershop@redusers.com

usershop.redusers.com



Visual Basic

Este libro está escrito para aquellos usuarios que quieran aprender a programar en VB.NET. Desde el IDE de programación hasta el desarrollo de aplicaciones del mundo real en la versión 2010 de Visual Studio, todo está contemplado para conocer en profundidad VB.NET al finalizar la lectura.

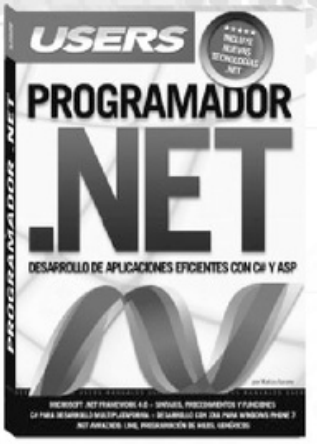
→ COLECCIÓN: MANUALES USERS
→ 352 páginas / 978-987-1773-57-2



Microcontroladores

Este manual es ideal para aquellos que quieran iniciarse en la programación de microcontroladores. A través de esta obra, podrán conocer los fundamentos de los sistemas digitales, aprender sobre los microcontroladores PIC 16F y 18F, hasta llegar a conectar los dispositivos de forma inalámbrica, entre muchos otros proyectos.

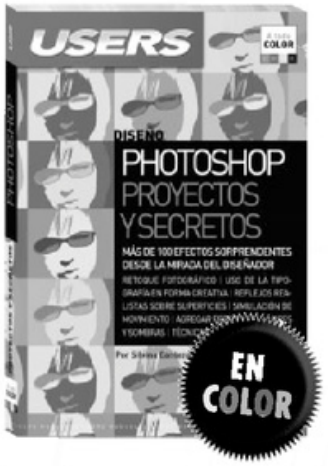
→ COLECCIÓN: MANUALES USERS
→ 320 páginas / 978-987-1773-56-5



Programador.NET

Este libro está dirigido a todos aquellos que quieran iniciarse en el desarrollo bajo lenguajes Microsoft. A través de los capítulos del manual, aprenderemos sobre POO y la programación con tecnologías .NET, su aplicación, cómo interactúan entre sí y de qué manera se desenvuelven con otras tecnologías existentes.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / 978-987-1773-26-8



Photoshop: proyectos y secretos

En esta obra aprenderemos a utilizar Photoshop, desde la original mirada de la autora. Con el foco puesto en la comunicación visual, a lo largo del libro adquiriremos conocimientos teóricos, al mismo tiempo que avanzaremos sobre la práctica, con todos los efectos y herramientas que ofrece el programa.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / 978-987-1773-25-1



WordPress

Este manual está dirigido a todos aquellos que quieran presentar sus contenidos o los de sus clientes a través de WordPress. En sus páginas el autor nos enseñará desde cómo llevar adelante la administración del blog hasta las posibilidades de interacción con las redes sociales.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / 978-987-1773-18-3



Administrador de servidores

Este libro es la puerta de acceso para ingresar en el apasionante mundo de los servidores. Aprenderemos desde los primeros pasos sobre la instalación, configuración, seguridad y virtualización; todo para cumplir el objetivo final de tener el control de los servidores en la palma de nuestras manos.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1773-19-0

¡Léalo antes Gratis!

En nuestro sitio, obtenga GRATIS un capítulo del libro de su elección antes de comprarlo.



Windows 7: Trucos y secretos

Este libro está dirigido a todos aquellos que quieran sacar el máximo provecho de Windows 7, las redes sociales y los dispositivos ultraportátiles del momento. A lo largo de sus páginas, el lector podrá adentrarse en estas tecnologías mediante trucos inéditos y consejos asombrosos.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1773-17-6



Desarrollo PHP + MySQL

Este libro presenta la fusión de dos de las herramientas más populares para el desarrollo de aplicaciones web de la actualidad: PHP y MySQL. En sus páginas, el autor nos enseñará las funciones del lenguaje, de modo de tener un acercamiento progresivo, y aplicar lo aprendido en nuestros propios desarrollos.

→ COLECCIÓN: MANUALES USERS
→ 432 páginas / ISBN 978-987-1773-16-9



Excel 2010

Este manual resulta ideal para quienes se inician en el uso de Excel, así como también para los usuarios que quieran conocer las nuevas herramientas que ofrece la versión 2010. La autora nos enseñará desde cómo ingresar y proteger datos hasta la forma de imprimir ahorrando papel y tiempo.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1773-15-2



Técnico Hardware

Esta obra es fundamental para ganar autonomía al momento de reparar la PC. Aprenderemos a diagnosticar y solucionar las fallas, así como a prevenirlas a través del mantenimiento adecuado, todo explicado en un lenguaje práctico y sencillo.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1773-14-5



PHP Avanzado

Este libro brinda todas las herramientas necesarias para acercar al trabajo diario del desarrollador los avances más importantes incorporados en PHP 6. En sus páginas, repasaremos todas las técnicas actuales para potenciar el desarrollo de sitios web.

→ COLECCIÓN: MANUALES USERS
→ 400 páginas / ISBN 978-987-1773-07-7



AutoCAD

Este manual nos presenta un recorrido exhaustivo por el programa más difundido en dibujo asistido por computadora a nivel mundial, en su versión 2010. En sus páginas, aprenderemos desde cómo trabajar con dibujos predeterminados hasta la realización de objetos 3D.

→ COLECCIÓN: MANUALES USERS
→ 384 páginas / ISBN 978-987-1773-06-0



Windows 7 Avanzado

Esta obra nos presenta un recorrido exhaustivo que nos permitirá acceder a un nuevo nivel de complejidad en el uso de Windows 7. Todas las herramientas son desarrolladas con el objetivo de acompañar al lector en el camino para ser un usuario experto.

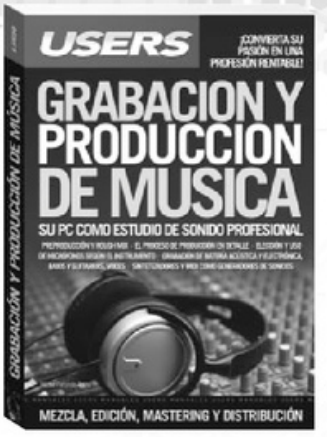
→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1773-08-4



Photoshop

En este libro aprenderemos sobre las más novedosas técnicas de edición de imágenes en Photoshop. El autor nos presenta de manera clara y práctica todos los conceptos necesarios, desde la captura digital hasta las más avanzadas técnicas de retoque.

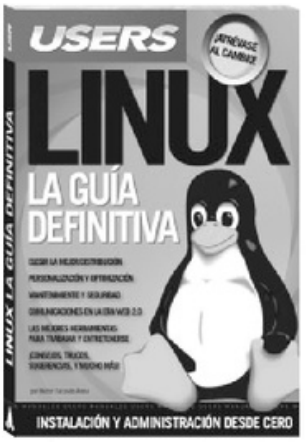
→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1773-05-3



Grabación y producción de música

En este libro repasaremos todos los aspectos del complejo mundo de la producción musical. Desde las cuestiones para tener en cuenta al momento de la composición, hasta la mezcla y el masterizado, así como la distribución final del producto.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1773-04-6



Linux

Este libro es una completa guía para migrar e iniciarse en el fascinante mundo del software libre. En su interior, el lector conocerá las características de Linux, desde su instalación hasta las opciones de entretenimiento, con todas las ventajas de seguridad que ofrece el sistema.

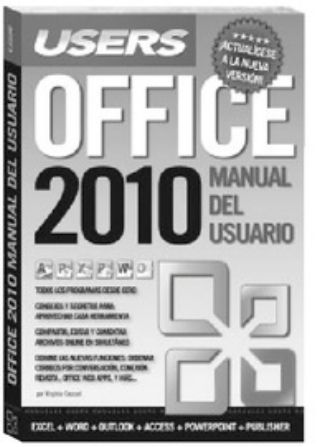
→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-26013-8-6



Premiere + After Effects

Esta obra nos presenta un recorrido detallado por las aplicaciones audiovisuales de Adobe: Premiere Pro, After Effects y Soundbooth. Todas las técnicas de los profesionales, desde la captura de video hasta la creación de efectos, explicadas de forma teórica y práctica.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-26013-9-3



Office 2010

En este libro aprenderemos a utilizar todas las aplicaciones de la suite, en su versión 2010. Además, su autora nos mostrará las novedades más importantes, desde los minigráficos de Excel hasta Office Web Apps, todo presentado en un libro único.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-26013-6-2

¡Léalo antes Gratis!

En nuestro sitio, obtenga GRATIS un capítulo del libro de su elección antes de comprarlo.



Excel Paso a Paso

En esta obra encontraremos una increíble selección de proyectos pensada para aprender, mediante la práctica, la forma de agilizar todas las tareas diarias. Todas las actividades son desarrolladas en procedimientos paso a paso de una manera didáctica y fácil de comprender.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-26013-4-8



C#

Este libro es un completo curso de programación con C# actualizado a la versión 4.0. Ideal tanto para quienes desean migrar a este potente lenguaje, como para quienes quieran aprender a programar desde cero en Visual Studio 2010.

→ COLECCIÓN: MANUALES USERS
→ 400 páginas / ISBN 978-987-26013-5-5



200 Respuestas: Seguridad

Esta obra es una guía básica que responde, en forma visual y práctica, a todas las preguntas que necesitamos contestar para conseguir un equipo seguro. Definiciones, consejos, claves y secretos, explicados de manera clara, sencilla y didáctica.

→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-26013-1-7



Funciones en Excel

Este libro es una guía práctica de uso y aplicación de todas las funciones de la planilla de cálculo de Microsoft. Desde las funciones de siempre hasta las más complejas, todas presentadas a través de ejemplos prácticos y reales.

→ COLECCIÓN: MANUALES USERS
→ 368 páginas / ISBN 978-987-26013-0-0



Proyectos con Windows 7

En esta obra aprenderemos cómo aprovechar al máximo todas las ventajas que ofrece la PC. Desde cómo participar en las redes sociales hasta las formas de montar una oficina virtual, todo presentado en 120 proyectos únicos.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-036-8



PHP 6

Este libro es un completo curso de programación en PHP en su versión 6.0. Un lenguaje que se destaca tanto por su versatilidad como por el respaldo de una amplia comunidad de desarrolladores, que lo convierten en un punto de partida ideal para quienes comienzan a programar.

→ COLECCIÓN: MANUALES USERS
→ 368 páginas / ISBN 978-987-663-039-9

usershop.redusers.com



200 Respuestas: Blogs

Esta obra es una completa guía que responde a las preguntas más frecuentes de la gente sobre la forma de publicación más poderosa de la Web 2.0. Definiciones, consejos, claves y secretos, explicados de manera clara, sencilla y didáctica.

→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-663-037-5



Hardware paso a paso

En este libro encontraremos una increíble selección de actividades que abarcan todos los aspectos del hardware. Desde la actualización de la PC hasta el overclocking de sus componentes, todo en una presentación nunca antes vista, realizada íntegramente con procedimientos paso a paso.

→ COLECCIÓN: PASO A PASO
→ 320 páginas / ISBN 978-987-663-034-4



200 Respuestas: Windows 7

Esta obra es una guía básica que responde, en forma visual y práctica, a todas las preguntas que necesitamos conocer para dominar la última versión del sistema operativo de Microsoft. Definiciones, consejos, claves y secretos, explicados de manera clara, sencilla y didáctica.

→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-663-035-1



Office paso a paso

Este libro presenta una increíble colección de proyectos basados en la suite de oficina más usada en el mundo. Todas las actividades son desarrolladas con procedimientos paso a paso de una manera didáctica y fácil de comprender.

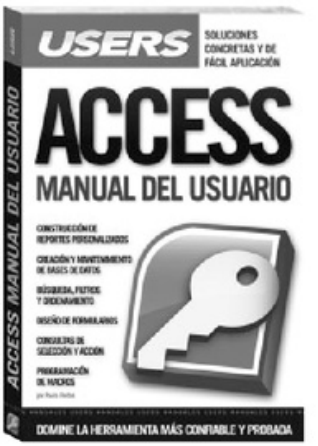
→ COLECCIÓN: PASO A PASO
→ 320 páginas / ISBN 978-987-663-030-6



101 Secretos de Hardware

Esta obra es la mejor guía visual y práctica sobre hardware del momento. En su interior encontraremos los consejos de los expertos sobre las nuevas tecnologías, las soluciones a los problemas más frecuentes, cómo hacer overclocking, modding, y muchos más trucos y secretos.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-029-0



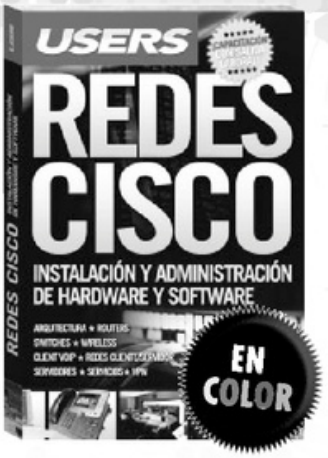
Access

Este manual nos introduce de lleno en el mundo de Access para aprender a crear y administrar bases de datos de forma profesional. Todos los secretos de una de las principales aplicaciones de Office, explicados de forma didáctica y sencilla.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-025-2

¡Léalo antes Gratis!

En nuestro sitio, obtenga GRATIS un capítulo del libro de su elección antes de comprarlo.



Redes Cisco

Este libro permitirá al lector adquirir todos los conocimientos necesarios para planificar, instalar y administrar redes de computadoras. Todas las tecnologías y servicios Cisco, desarrollados de manera visual y práctica en una obra única.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-024-5



Proyectos con Office

Esta obra nos enseña a usar las principales herramientas de Office a través de proyectos didácticos y útiles. En cada capítulo encontraremos la mejor manera de llevar adelante todas las actividades del hogar, la escuela y el trabajo.

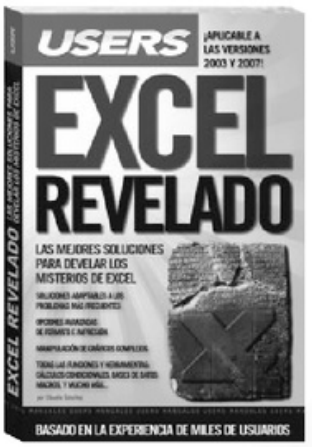
→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-023-8



Dreamweaver y Fireworks

Esta obra nos presenta las dos herramientas más poderosas para la creación de sitios web profesionales de la actualidad. A través de procedimientos paso a paso, nos muestra cómo armar un sitio real con Dreamweaver y Fireworks sin necesidad de conocimientos previos.

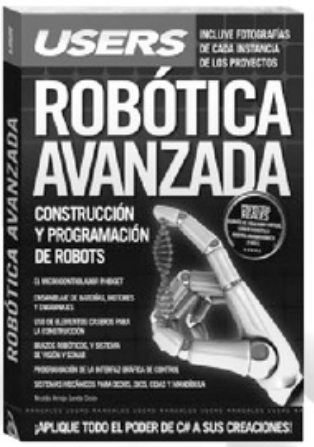
→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-022-1



Excel revelado

Este manual contiene una selección de más de 150 consultas de usuarios de Excel y todas las respuestas de Claudio Sánchez, un reconocido experto en la famosa planilla de cálculo. Todos los problemas encuentran su solución en esta obra imperdible.

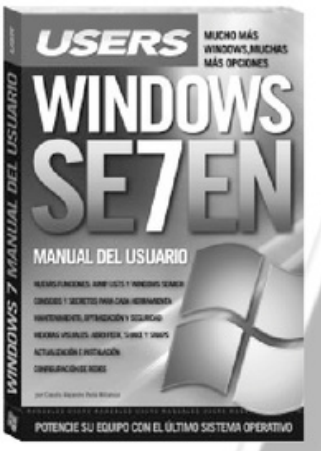
→ COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-021-4



Robótica avanzada

Esta obra nos permitirá ingresar al fascinante mundo de la robótica. Desde el ensamblaje de las partes hasta su puesta en marcha, todo el proceso está expuesto de forma didáctica y sencilla para así crear nuestros propios robots avanzados.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-020-7



Windows 7

En este libro, encontraremos las claves y los secretos destinados a optimizar el uso de nuestra PC tanto en el trabajo como en el hogar. Aprenderemos a llevar adelante una instalación exitosa y a utilizar todas las nuevas herramientas que incluye esta versión.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-015-3

usershop@redusers.com

DESARROLLO DE ÚLTIMA GENERACIÓN CON UN LENGUAJE LIBRE Y FLEXIBLE



Este manual presenta Ruby, uno de los lenguajes de programación más flexibles y poderosos de la actualidad para el desarrollo de aplicaciones de escritorio o Web. Por su sencillez, es ideal para dominarlo rápidamente, aun sin tener grandes conocimientos de programación.

» DESARROLLO
» 432 PÁGINAS
» ISBN 978-987-1347-67-4





LLEGAMOS A TODO EL MUNDO VÍA »OCA * Y DHL **

* SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

usershop.redusers.com // usershop@redusers.com

USERS

JAVA

DOMINE EL LENGUAJE LÍDER EN APLICACIONES CLIENTE-SERVIDOR

CONTENIDO

1 | PROGRAMACIÓN ORIENTADA A OBJETOS
Historia / Conceptos / Terminología / Polimorfismo / Tipos y subtipos / Herencia / ¿Por qué objetos?

2 | INICIACIÓN A JAVA
Historia / Preparación / Eclipse IDE / Test Driven Development / Primeros códigos

3 | SINTAXIS
Palabras clave / Ciclos / Declaraciones, expresiones, sentencias y bloques / Otras estructuras / Tipos primitivos y literales / Operadores / Paquetes

4 | CLASES
Definición / Atributos / Métodos / La herencia y los métodos / Constructores / This y Super / Estático vs. no estático

5 | MÁS CLASES
Clases abstractas / Clases anidadas / Clases anidadas estáticas / Clases internas / Clases locales y clases anónimas / Ejercicio: el juego de la vida

6 | INTERFACES
Definición / Uso / Clases abstractas vs. interfaces

7 | ENUMERACIONES
Definición / Uso

8 | EXCEPCIONES
Definición / Uso / Excepciones chequeadas / Excepciones no chequeadas / Errores

9 | GENÉRICOS
Definición / Subtipado / Comodín / Tipos restringidos / Genéricos en el alcance estático

10 | LIBRERÍA BASE
Librería y objetos básicos / Colecciones / Clases útiles / I/O

11 | ANOTACIONES
¿Qué son las anotaciones? / Definición / Uso de las anotaciones / Acceder a las anotaciones en tiempo de ejecución / Distintos usos de las anotaciones

12 | TÉCNICAS Y DISEÑO
Java Beans / Inyección de dependencias e inversión de control / Evitar null / Representar conceptos con objetos

NIVEL DE USUARIO

PRINCIPIANTE | INTERMEDIO | AVANZADO | EXPERTO

Este libro fue concebido para ayudar a quienes buscan aprender a programar en Java, como así también para aquellos que conocen el lenguaje, pero quieren profundizar sus conocimientos.

Con la lectura de esta obra, iniciaremos un apasionante y progresivo recorrido, que comenzará con los conceptos fundamentales de la programación orientada a objetos, el diseño y el desarrollo de software. El aprendizaje será completo desde el momento en que tendremos en cuenta las buenas prácticas y las mejores técnicas para obtener programas robustos, claros y eficientes.

Todos los procedimientos son expuestos de manera práctica con el código fuente de ejemplo (disponible en Internet), diagramas conceptuales y la teoría necesaria para comprender en profundidad cada tema presentado. Al finalizar el libro, el lector contará con las herramientas requeridas para ser un programador integral en Java.

Ignacio Vivona es un desarrollador de software con más de diez años de experiencia y un entusiasta de la tecnología desde siempre. Gracias a su guía y consejos, estaremos en condiciones de conocer a fondo el lenguaje y ponerlo en práctica para insertarnos en el mercado del desarrollo de software.

RedUSERS.com

En este sitio encontrará una gran variedad de recursos y software relacionado, que le servirán como complemento al contenido del libro. Además, tendrá la posibilidad de estar en contacto con los editores, y de participar del foro de lectores, en donde podrá intercambiar opiniones y experiencias.

Si desea más información sobre el libro puede comunicarse con nuestro Servicio de Atención al Lector: usershop@redusers.com

JAVA

This book is the key to access the world of programming with Java, the leading language in client-server applications. Know all the secrets of this object-oriented programming language, the most widely used today by developers.

ISBN 978-987-1773-97-8

MANUALES USERS MANUALES USERS MANUAL

DESARROLLO PROFESIONAL MULTIPLATAFORMA